THE CONSORTIUM REQUIREMENTS ENGINEERING METHOD

SPC-92118-CMC

MDA972-92-J-1018

VERSION 01.00.00

SEPTEMBER 1991

THE CONSORTIUM REQUIREMENTS ENGINEERING METHOD

SPC-92118-CMC

VERSION 01.00.00

SEPTEMBER 1991

Stuart Faulk
James Kirby, Jr.
Skip Osborne
D. Douglas Smith
Steven Wartik

John Brackett, Boston University

Paul T. Ward, Software Development Concepts

Reprinted for the
VIRGINIA CENTER OF EXCELLENCE
FOR SOFTWARE REUSE AND TECHNOLOGY TRANSFER

February 1993

SOFTWARE PRODUCTIVITY CONSORTIUM, INC.
SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070

Macintosh is a registered trademark of Apple Computer, Inc.

SuperCard is a copyright of Silicon Beach Software, Inc.

team*work* is a registered trademark of Cadre Technologies, Inc.

CONTENTS

FIGURES

This page intentionally left blank.

TABLES

PREFACE

The Consortium Requirements Project was created to address the key member company problem that:

> Requirements are incomplete, misunderstood, poorly defined, and change in ways that are difficult to manage. (Consortium Board of Directors).

Work in 1990 determined that these problems could be substantially addressed by providing improved methods and tools targeted to the needs of line projects. The first goal of the project was to understand, in detail, what problems line projects have with requirements and where current methods or practices are deficient. This produced a detailed set of requirements for methods and tools supporting the requirements process. The second goal of the project was to develop a method and supporting tool that satisfied these member company requirements. This work initially focused on software (as opposed to system) requirements. Subsequent work will seek to validate the method through pilot projects in member companies. Validated products will be transferred to line projects through detailed guidebooks, examples, and training.

To ensure timely feedback to the member companies, the Consortium will provide reports over the course of the project. This is the first such report describing the requirements method. Its purpose is to permit member company personnel to assess the project against its goals, i.e., the method requirements established by the member companies. The Consortium with will use the feedback to refine the statement of method requirements and make course corrections in the technical program as necessary. The report is also intended to initiate interest in a pilot project that will apply the method experimentally in a member company setting (e.g., an Internal Research and Development [IR&D] effort) in 1992.

This page intentionally left blank.

ACKNOWLEDGEMENTS

The Consortium wishes to thank the attendees of the 1991 Requirements Workshop for their work on the method requirements, the members of the Technical Advisory Group for refining the method requirements and for reviewing this report, and to Randy Scott for reviewing this report. Special thanks also go to Paul Clements and Jim O'Connor for many helpful comments and suggestions.

Parts of this report, particularly the example (Appendix B), come from (van Schouwen 1990).

This page intentionally left blank.

EXECUTIVE SUMMARY

The Consortium Requirements Project was created to address the key member company problem that:

> Requirements are incomplete, misunderstood, poorly defined, and change in ways that are difficult to manage (Consortium Board of Directors).

Preliminary investigation in 1990 determined that the Consortium could substantially address these problems by providing improved methods and tools targeted to the needs of line projects. The first goal of the project was to understand, in detail, what problems line projects have in developing requirements and where current methods or practices are deficient. This phase of the project produced a detailed set of requirements for methods and tools supporting the software requirements process. The second goal of the project was to develop a method and supporting tool that satisfied member company needs. This work, at the behest of member companies initially focused on software (as opposed to system) requirements. This is the first report on the technical approach and results.

The project first focused on assimilating existing technology, inventing new technology only where necessary. This ensured that time was not wasted in developing new techniques or tools where suitable technology was already available. The goal was to create a single method that incorporated the best of a few proven technologies, the choice of technologies being driven by the member company needs.

The first cycle of method development described in this report focused on integrating two key technologies that, together, address most of the high-priority member company needs: Paul Ward's CASE Real-Time Method and work by David Parnas and colleagues on embedded system requirements. The resulting method has the following features:

- Modeling and specification of real-time systems.
- Object orientation providing for separation of concerns, locality, and concurrent development.
- Management of fuzzy or changeable requirements.
- Equivalent graphic and textual specification using existing notations.
- Document- and standard-independent behavioral and data models.
- Precise, testable, and nonalgorithmic specification.
- Applicability with existing tools.
- Formal basis suitable for automation and eventual formal validation.

To the extent that the Consortium has validated the proposed method with small examples, the approach appears sound. It has achieved the initial technical goal of merging a strong, graphic-based method with a formal text-based method with the desired result of exploiting the best features of each approach. It has successfully applied the method to a small, real-time problem with results that are consistent with the high-priority member needs.

A variety of needs remain to be addressed. These include specific mechanisms to handle traceability, the extension to command and control applications, and guidelines for mapping the method and its products to specific standards. Of particular concern are issues of scale and integration with the system requirements process. The work to date confirms the need to address at least some aspects of the system requirements problem to fully address open issues in software requirements.

Work in the next development cycle will concentrate on the remaining member company needs. Work in the remainder of this year will focus on completing the first cycle of tool prototyping and on arranging a suitable pilot application of the method in an IR&D effort. The pilot application and supporting method development in 1992 will focus on the remaining needs and provide more realistic examples. The Consortium will address all of the critical issues necessary to a practicable method by the end of 1992. A complete guidebook on the method is scheduled for 1993.

1. INTRODUCTION

The Consortium's Requirements Engineering (CoRE) method is being developed in response to the member companies' need for improved requirements technology. Prior work by the Requirements Project and member company personnel produced a detailed statement of needs. This report describes the technical progress in meeting those needs. It gives an overview of CoRE and illustrates CoRE with a small example. It also discusses the technical rationale for the method in terms of the stated member company needs.

Because the report describes ongoing work, the methods discussed are necessarily incomplete. While parts of CoRE are well-defined, several issues remain open. In particular, the Consortium has not yet fully addressed global issues relating to the overall system development process and scale-up. For this reason, the focus of this report is on conveying an intuitive understanding of the technical approach rather than on a detailed technical description of CoRE. The report is intended to provide information about the development of CoRE that the reader can use to assess progress against the member company needs. Goals of the report include initiating feedback from the members on the technical content and stimulating interest in a pilot (IR&D) project applying CoRE.

1.1 APPROACH TO METHOD DEVELOPMENT

Two key principles drive the Consortium's method development approach: customer orientation and use of available technology. To develop a methodology (or tool) that is usable by line projects, addresses their problems with requirements, and represents a substantial improvement over available methods, there must be an ongoing dialogue between method developers and method users. For this reason, there is a continuing effort in the Requirements Project to understand and capture member company needs and to evaluate its products against those needs. The project initially captured these needs in a set of requirements that CoRE must satisfy.

The method requirements drive the project's technology acquisition. A basic assumption of the method development approach is that there is technology available that addresses the bulk of member company requirements. While no single methodology may satisfy all requirements, the best parts of a small number of methodologies will suffice. Thus, the approach is to acquire and integrate existing methods using the method requirements to evaluate the suitability of candidate technologies. The value added by the project includes integration of the component methods into a coherent whole, case studies to demonstrate applicability in member company environments, detailed guidebooks and examples targeted to project personnel, and prototype tool support. It will also include tools directly supporting the advanced features of CoRE.

Work on CoRE itself has followed a development strategy of rapid prototyping and reuse. The project first focused on assimilating existing technology, inventing new technology only where necessary. This ensures that time is not wasted in developing new techniques or tools where existing technology

satisfies the method requirements. The goal is to create a single method that incorporates the best of a few proven technologies. The project chose this approach because prior work showed that, while no single available method meets all the member needs, there are available methods that possess complementary strengths with respect to the method requirements. Further, these methods are based on common technical assumptions. While these methods differ in process and notation, they agree on **what** information must be captured and on the underlying technologies appropriate to capture it.

The project used the rapid prototyping strategy for risk mitigation and incremental validation. This strategy included not only the creation of prototype tools supporting CoRE, but rapid cycles of development and application of CoRE itself. A few steps or features were added to CoRE at one time, then tried out on a sample problem. This allowed a quick determination of whether proposed approaches addressed the method requirements as intended.

Since this report describes ongoing work, the methods discussed vary in their relative maturity. It addresses the basic approach, underlying formal models, and supporting technologies in some detail. However, it addresses broader life-cycle issues such as the overall development process and issues of scale-up only superficially. Since the example problem (Appendix B) illustrating CoRE was also produced under these constraints, it represents a mix of technologies at different levels of development. Subsequent efforts will increase the scale of the examples with pilot projects, providing more realistic examples.

1.2 THE METHOD REQUIREMENTS

The Requirements Project determined the qualities CoRE must have from guidance provided by the Technical Advisory Board (TAB), the Requirements Workshop (Faulk et al. 1991), and the Requirements Technical Advisory Group (TAG). In discussions with the TAG, the method requirements produced by the workshop were consolidated and an agreement was made concerning which needs should be addressed in the initial development cycle covered by this report. The order follows the priorities set by the workshop.

The remainder of this section provides an overview of the method requirements. The requirements addressed by the current work are given first in approximate order of priority. Then the open critical requirements are listed. These will be addressed over the remainder of 1991 and by work in 1992.

1.2.1 Requirements Addressed in the Current Method

The current CoRE method addresses the following requirements:

- *Member Company Applications.* CoRE must support the development of precise, testable specifications for real-time/embedded systems.
- *Changing Requirements.* CoRE must support developing requirements specifications that are easy to change throughout the software life cycle. When requirements change, it must be easy to tell what parts of the requirements specification and other work products are affected.
- *Audience.* CoRE must support the development of requirements specifications that are understandable and useful to the specification's entire audience. The notation used by those applying CoRE must be understandable to systems engineers, hardware engineers, and software

staff. CoRE should use familiar abstractions that eliminate the need to understand underlying formalisms. It must support the ability to present customer-oriented views of system requirements.

- *System Interface.* CoRE must provide mechanisms that allow the description of system boundaries and the precise definition of system interfaces. It must provide mechanisms that allow the description of both a system and the environment in which it operates. The environment may include other systems that are under development.

- *Document and Format Independence.* CoRE must include principles, guidelines, and techniques for representing requirements independent of particular documents or work products.

- *Separation of Concerns.* CoRE must support the definition of requirements as a set of distinct and relatively independent parts. It should support localizing requirements that are fuzzy or incomplete and allow work to proceed where requirements are well understood.

- *Derivation from System Requirements.* CoRE must include guidelines and examples of required inputs for the software requirements process and the form such inputs must take.

- *Nonalgorithmic Specification.* CoRE should allow nonalgorithmic specification of requirements (where an algorithm is not actually required).

- *Consistent Requirements.* CoRE must define what makes a set of requirements consistent (unambiguous). It must include principles, guidelines, and techniques for determining whether requirements are internally consistent and for keeping them internally consistent after changes have been made to them.

- *Complete Requirements.* It must be possible for users of CoRE to determine where the requirements specification is internally incomplete. It must permit definition and use of incomplete requirements. For example, CoRE must allow detailing and checking of one part of the requirements before another is complete.

1.2.2 Needs Remaining to be Addressed

In this initial method development cycle, the CoRE method did not address or only partially address a number of aspects. In most cases these represented method requirements for which a solution depends on the choices made in developing CoRE. For instance, most process issues cannot be addressed in detail until the scope and content of the work products are determined. Similarly, the relation to other life-cycle work products (e.g., traceability to system requirements or the mapping to design) cannot be effectively addressed until the requirements method is reasonably well-defined. This section gives an overview of the requirements remaining to be addressed in subsequent method development cycles.

- *Command and Control.* CoRE must support specification of requirements for command and control systems.

- *Traceability.* CoRE must support tracing between components of a requirements specification and components of other work products, including design specifications and test plans.

- ***Specification of All Requirements.*** CoRE must support the development of specifications of functional requirements and nonfunctional requirements (it currently supports functional but does not directly address some nonfunctional requirements).

- ***Process.*** CoRE must be amenable to waterfall and nonwaterfall software processes. It must be tailorable specifically to the Consortium's Evolutionary Spiral Process and to processes that involve prototyping. It must address where in the process executable requirements models are appropriate, and it must support evolution to the Synthesis process.

- ***Life Cycle.*** CoRE must provide guidelines, strategies, and techniques for interfacing with other activities in the software life cycle. It should provide to systems engineering a definition of the inputs from systems engineering (and the form the inputs must take) required for specifying software requirements. It must be possible to transition to commonly used design methods. Transition to Consortium design methods must be well-defined.

- ***Standards.*** CoRE must support adherence to selected current and emerging standards. CoRE must provide guidance on producing requirements work products that are acceptable to DOD-STD-2167A.

- ***User Interface Requirements.*** CoRE must include principles, guidelines, and techniques for specifying user interface requirements. It should be possible to generate prototypes of the interface from the specification.

- ***Multiple Views of Requirements.*** CoRE must specify how to convert from an object-oriented representation of requirements to a functional representation and back.

1.3 KEY ASSUMPTIONS AND FEATURES OF THE METHOD

The method requirements are manifested in CoRE in two ways: first, in the basic set of assumptions the method makes about the requirements process and its products; second, in determining which existing technologies are suitable candidates for incorporation in CoRE. These are treated briefly here and in more detail in Section 5.

1.3.1 Method Assumptions

The basic assumptions of the CoRE method are:

- ***Reactive System Orientation.*** Member company needs focus on methods for real-time embedded systems and command and control. The project's initial work concentrates on real-time embedded systems. The work assumes some characteristics typical of such systems such as the fact that these systems typically maintain ongoing relationships with the environment. Because such systems must react to environmental changes, they are sometimes called **reactive** systems (Harel and Pnueli 1985). Subsequent work will extend CoRE to command and control applications.

- ***Front Loading.*** Formal studies (e.g., by Boehm [Boehm 1981]) and member company experience confirm that the early development phases have the highest cost leverage. The approach calls for front loading in the sense that sufficient resources will be expended on the

requirements phase, and on subsequent maintenance and refinement of requirements, to develop a high-quality requirements specification that will serve the many users of software requirements throughout the life cycle.

- ***Document Independence.*** CoRE will be practiced in many member environments with different internal standards for documentation. CoRE should not assume anything about the way the final product is documented except that it must be possible to use a variety of formats including DOD-STD-2167A.

- **Nonalgorithmic Specification.** CoRE must be able to express all behavioral requirements of system software without mandating a particular design or implementation (although these must be **allowed** by CoRE if such are actual requirements). The underlying model must support such nonalgorithmic specification.

- ***Existing Notations and Tools.*** The cost of training and tools support dictates the use of existing resources. The approach to developing CoRE assumes that existing notations and tools should be incorporated wherever possible to reduce the cost of acquiring and using CoRE.

1.3.2 Technical Features

The technical features of the CoRE method are:

- ***Graphics and Text.*** The ability to quickly develop and communicate system overviews was determined a key capability as was the ability to precisely express detailed requirements. The first requirement is better supported by graphic-based methods where the second is better addressed by text-based approaches. The method integrates technology from each camp, with equivalent semantics where the graphical and textual representations overlap.

- ***Object Orientation.*** Key needs include scaleability, ease of change, and the ability to separate concerns. Object-based models provide the necessary encapsulation and abstraction to address these requirements so that the method uses an object model in its definition and organization. The object model also addresses concerns for transition to designs, especially in Ada.

- ***Machine-Based Model.*** The approach uses an underlying model based on expressing requirements in terms of concurrent state machines. This addresses member concerns that extensive training in formal methods not be required to apply the method or understand the specifications produced. It also addresses concerns for the ability to perform nontrivial analysis of properties like completeness and consistency. While having many of the analytic virtues of formal methods, the approach is generally simpler and more intuitive to use than methods that express requirements entirely in statements of a formal logic. Formal logic can still be exploited freely behind the scenes to build tools that aid in requirements analysis.

1.4 TECHNICAL APPROACH TO THE METHOD

Technical work in this first increment of method development focused on combining key technologies from two methods, Paul Ward's CASE Real-Time Method (Ward 1989) and work by David Parnas and others (Parnas and Madey 1990) on embedded system requirements. These were chosen as the first components of the method because:

- The two approaches shared a common set of assumptions about what information must be captured in requirements.
- The approaches used much of the same technology (e.g., finite state machines, functions) to specify requirements.
- The two approaches had complementary strengths, each addressing some critical needs.
- Together, the two methods addressed all of the key requirements that the method is scheduled to satisfy in the first increment.

The proposed method currently uses a front end based on Ward's object-modeling techniques and a back end based on Parnas' formal methods. Early phases of understanding system context, problem analysis, problem modeling, and requirements organization were taken from Ward's object-oriented model for requirements analysis. The graphic approach to static and dynamic system modeling employing existing notations were also based on Ward's work. This approach allowed any of a set of commonly available graphic notations to be used so commonly available CASE tools could be exploited. The characteristics of the behavior model were based on Parnas's four-variable model for embedded system requirements. Techniques for capturing detailed requirements in the behavioral model, timing and accuracy requirements, and methods for analysis of completeness and consistency were also based on the work of Parnas and his colleagues. These complementary methods were integrated in the overall process described in Section 3.

Subsequent work will seek to integrate other promising technology for formal reasoning about requirements, execution of high-level specifications, and automated support for assessing completeness, consistency, timing constraints, and safety properties.

1.5 REPORT ORGANIZATION

Section 2, The Behavioral and Data Models, gives an overview of the standard behavioral model and the document-independent data model that underlies CoRE. It describes the underlying formalisms, the essential information in a specification developed using the model, and how pieces of information are related. It is the qualities of these models that determine the character of CoRE, so this section should be read before reading about CoRE itself.

Section 3, The Conceptual Method, provides an overview of the analysis and specification approach based on the behavioral model. It gives an overview of the requirements process envisioned for CoRE and discusses how CoRE is applied to develop a requirements specification. The steps and methods of specification are discussed in some detail in terms of the behavioral model and document-independent data model. This part of the discussion does not assume any particular notation, documentation standard, or format.

Section 4, Methods of Representation, discusses a variety of methods for organizing and representing requirements. These include graphic notations, specification templates, and specific techniques for specifying state machines and functions. This work is included for several reasons. First, to actually try CoRE and produce examples, particular methods of representing requirements specification had to be chosen. This discussion helps familiarize the reader with the conventions and notations so the examples can be fully understood. Second, the method requirements specify existing notation be exploited. This section describes how that is accomplished. Finally, since the choice of notation is important, the section discusses the rationale behind particular choices.

Section 5, The Technical Rationale and Progress, answers the questions, How are the member company needs addressed by CoRE? and Why has a particular technique or approach been used in the method? It discusses **why** the Consortium chose the component technologies and encompassing methods in terms of how they satisfy the method requirements. For instance, the section discusses the reasons the technologies used in CoRE support development of complete or maintainable requirements. It also describes how much progress has been made in addressing the method requirements defined by the member companies.

Section 6, Conclusions, summarizes the state of CoRE, the method requirements that CoRE does not yet address, and what the project plans to do in 1992 and 1993.

The appendixes provide an example requirements specification and discussion.

Appendix A provides a walk-through of the specification from the point of view of different users. It should be read before or in conjunction with the example (Appendix B). This section conveys how the benefits of CoRE are realized in using the product (e.g., for maintaining requirements or developing scenarios and test cases) by showing how different users would use the specification to get an overview of the system or answer detailed questions about required behavior.

Appendix B contains a detailed specification of a small, real-time system, the Fuel-Level Monitoring System (FLMS). The example illustrates the basic approach and features of the method from the top-level decomposition into objects to the detailed specification of the system inputs and outputs. The example is necessarily small and does not address issues of scale. These will be addressed in subsequent examples that will be provided as addendums to this report.

For the reader seeking to understand CoRE and its rationale, the report should be read in order. Readers interested only in understanding CoRE itself can skip Section 5.

1.6 TYPOGRAPHIC CONVENTIONS

This report uses the following typographic conventions:

Serif font ..... General presentation of information.

Italicized serif font ..... Publication titles.

Boldfaced serif font ..... Section headings and emphasis.

This page intentionally left blank.

2. THE BEHAVIORAL AND DATA MODELS

Much of the leverage of the Consortium's approach comes from standardizing the underlying models used to capture behavioral requirements and organize requirements information. CoRE is object-oriented in the sense that objects are the primary architectural component used to structure requirements information. The approach differs from many of the object-oriented methods in that a standard, formal model of embedded system behavior (called the four-variable model) guides the object decomposition and definitions. The behavioral model defines what kinds of information can be provided by an object interface and used by other objects. It also determines what information must be included in the object definitions. The use of the standardized model (behavioral data) allows the method to provide explicit guidance to the developer in choosing an appropriate set of objects, in writing the object definitions, and in deciding when the specification is complete and consistent.

Where the behavioral model determines what information must be captured, the data model describes how the information is organized. It formalizes the contents of a requirements specification in terms of a document-independent model of the information. The developer can then map the contents of the data model to different document formats, depending on member company needs.

The remainder of this section describes the two models. Since this is an introductory report and the work is yet incomplete, the emphasis is on conveying an intuitive understanding. Subsequent work will further refine and formalize the model. Subsequent reports will provide more formal, detailed, and complete descriptions. The discussion of the behavioral model itself is, in part, summarized from works by Parnas (Parnas and Madey 1990) and van Schouwen (van Schouwen 1990). A more formal discussion of the behavioral model and its mathematical basis can be found in these and the other works by Parnas and others.

2.1 THE STANDARD BEHAVIORAL MODEL

While CoRE concentrates on software requirements, the originating technology addresses both system and software requirements. The originating technology assumes that any system to be built exists within, and interacts with, a particular environment. The nature of the application dictates which aspects of the environment are of interest. For instance, an engine control system needs information about the ambient air pressure but not current altitude, while an aircraft control system requires information about the altitude, the barometric pressure being only a means to the desired quantity. These physical quantities of interest are called the environmental variables and are represented in the behavioral model with mathematical variables "in the way that is usual in engineering" (Parnas and Madey 1990). Constraints exist between the environmental variables due to the properties of the physical world (e.g., force of gravity), the physical characteristics of the overall system (e.g., the maximum rate of climb of an aircraft), or the characteristics of other systems in the environment. The system itself, since it affects the environment, will introduce other constraints on the environmental variables.

Refinement of the system specification to a software specification requires introducing additional constraints, i.e., those corresponding to the required behavior. The system design determines which constraints on the environmental variables are maintained by the software and which are maintained by the hardware. The allocation of certain system constraints to the hardware creates additional environmental constraints for the software. The system design process also determines what computing and other resources the software will have available. These choices determine the precision with which the software can determine or affect the values of environmental variables. Figure 1 (van Schouwen 1990) shows the standard model of the system interacting with its environment.

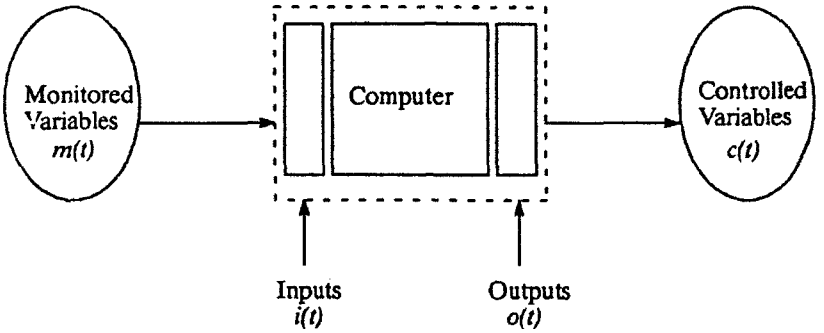

Figure 1. Standard Embedded System Model

From the point of view of the software requirements, the environmental variables are monitored, controlled, or both. The monitored variables represent quantities in the environment that the system must track. Controlled variables represent the things in the environment the system controls (e.g., a valve, display, or firing mechanism).

Engineers capture the required software behavior as a set of relations among the values of monitored and controlled variables. This corresponds to the intuitive notion of required behavior in that it relates specific, observable changes in the environment (e.g., the trigger is pressed) to observable system actions (e.g., the weapon is released). In defining the externally visible behavior, there are two relations of interest, called NAT (for NATure) and REQ (for REQuired). The NAT relation describes those constraints placed on the system by the external environment (e.g., physical laws and the properties of physical systems). When talking about software requirements, NAT includes the properties of the interfacing hardware. The REQ relation describes the required system behavior. Engineers complete the specification is completed by specifying the values provided by the system's hardware devices (or, if necessary, suitable abstractions of those values) called the input data items and output data items. These describe the available hardware resources control and monitor how the environmental variables. This is expressed in two additional relations called IN (for INput) and OUT (for OUTput). Figure 2 shows the variables and relations.

2.1.1 Relations NAT and REQ

To specify the required behavior, engineers represent each environmental variable by a mathematical variable and clearly define the association between the physical quantity and the mathematical variable. Each such variable then has a value that is a function of time. For this discussion, the function giving the values of the set of all monitored variables over time is denoted as $m(t)$ and that giving the values of all controlled variables is denoted as $c(t)$. Then, the relations NAT and REQ define the constraints

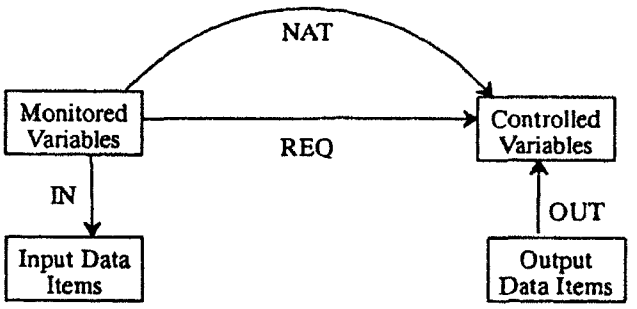

Figure 2. The Requirements Relations

on the environmental variables determined by external factors (e.g., physical laws) and those the system is required to maintain, respectively, where each is a relation from *m(t)* into *c(t)*. In summary:

- ***Relation NAT.*** The external environment and other systems constrain the possible values of the environmental variables; for instance, the rate at which the fuel level in a tank can change when a pump is turned on or the maximum rate of change of the temperature in a reaction vessel. NAT is defined as follows:
  - NAT is a relation from *m(t)* into *c(t)*.
  - The ordered pair (*m(t)*, *c(t)*) is in NAT only if the controlled variables can take the values described by *c(t)* when the monitored variables have the values given by *m(t)*.
- ***Relation REQ.*** The computer system is intended to impose additional constraints on the values of the controlled variables. These constraints are what are typically thought of as the functional requirements. For instance, the heater is required to be on if the temperature in the reaction vessel falls below 500 degrees (equivalently, and more formally, the controlled variable that is the heater state is constrained to have the value "on" whenever the state of the environment is such that the monitored variable corresponding to the vessel temperature has a value less than 500 degrees). REQ is then defined as:
  - REQ is a relation from *m(t)* into *c(t)*.
  - The ordered pair (*m(t)*, *c(t)*) is in REQ if and only if the computer system may permit the controlled variables to take on the values given by *c(t)* whenever the monitored variables have the values given by *m(t)*. The implementation must take on a subset of the values allowed by the relation so that there is a controlled variable value corresponding to every possible state of the monitored variables.

Typically, NAT is a relation rather than a function since there are many possible values of the controlled variables for a given environmental state. REQ is a relation rather than a function because there is typically tolerance in the required behavior. That is, the system must display the temperature to within plus or minus 0.05 degrees, so there are a set of possible values the system could display for a given temperature and still satisfy the requirements.

2.1.2 Relations IN and OUT

Ultimately, the system design process must identify the resources available to the software to determine the values of the monitored variables and affect the values of controlled variables. Early

in the process, engineers may represent them by abstractions of the ultimate inputs and outputs. When the hardware becomes defined, the interface devices provide these values. In any case, where the software is required to do its job using certain inputs and outputs, the software requirements must reflect these inputs and outputs. The model captures them with two additional relations, one giving the correspondence between the monitored variables and the system inputs over time *i(t)*, and the other giving the correspondence between the system outputs over time *o(t)* and the controlled variables:

- ***Relation IN.*** Relation IN describes what the software will see in terms of the available inputs for possible values of the monitored variables. This specifies the accuracy with which the system can measure the environmental values of interest. More precisely, IN is defined as:
  - IN is a relation from *m(t)* into *i(t)*.
  - The ordered pair (*m(t)*, *i(t)*) is in IN if and only if *i(t)* gives the possible values of the inputs when the monitored variables have the values given by *m(t)*.
- ***Relation OUT.*** Relation OUT specifies (mathematically) how the controlled variables are affected by sending particular values to the output devices. OUT is defined as:
  - A relation from *o(t)* into *c(t)*.
  - The ordered pair (*o(t)*, *c(t)*) is in OUT if and only if the controlled variables will take on the values given by *c(t)* whenever the outputs are assigned the values given by *o(t)*.

IN is a relation since input devices have limited accuracy. Similarly, OUT is a relation because output devices have limited precision. Both input and output devices have associated delays.

2.1.3 Object Decomposition and Definition

The requirements information specified according to the behavioral model is structured as a set of communicating objects. The key feature of CoRE is that it defines all the objects and the information communicated between objects in terms of the behavioral model. As in all object-oriented methods, each object consists of a public part that other objects can use (called the object interface) and a private part that cannot be used outside the object definition itself (called the encapsulated or hidden information). The behavioral model constrains the contents of the object definitions, the types of information that must be hidden, and the types of information that can be provided on a object interface. In particular, the engineers must express all of the public information provided by the objects in terms of the environmental variables. Thus, the object interfaces only provide information in terms of the state of environmental variables, changes in their state, or the history of such changes. This keeps the behavioral model consistent with the goal of providing nonalgorithmic specification. The object definitions encapsulate information that is likely to change, such as the details of the hardware interfaces. The data model in Section 5 illustrates this in more detail. Briefly:

- ***Monitored and Controlled Variables.*** The engineers allocate definitions of monitored and controlled variables among the system interface objects. They specify these as part of the object's interface to the external environment. Every monitored variable must be input to at least one object, and exactly one object contains the variable definition. Every controlled variable is the output of exactly one object; that object encapsulates its specification. The system interface objects, in general, hide the details of the environmental variable specifications.

- ***IN and OUT Relations.*** Details concerning how the value of a monitored variable is read from the hardware or a controlled variable is set must be private to some object in the system. Where there is a simple correspondence between the input/output data items and the environmental variables (as there is in the FLMS example), the object that encapsulates the environmental variable definition also encapsulates the corresponding data item definition and IN or OUT relation specification.

- ***Shared State Information.*** Decomposition of the specification into objects means that different information about the system state are specified in different objects. Where engineers must use this information to define other objects (e.g., in the specification of the output functions), they define it on the object interface. Engineers provide this information in the form of terms, events, or modes.

  A term describes some information about the system state, such as the current state of some monitored variable. The behavioral model requires that the engineers write all terms as functions of the monitored or controlled variables.

  An event denotes the instant at which there is a change in the state of one or more environmental variables. Events are always functions of the monitored variable time (and may be functions of other variables in addition).

  Information about the system history must also be shared among objects. Modes, classes of system states, capture system history. Objects (called mode class objects) encapsulate the details of which events cause which state transitions (also called control objects in some methods). The mode class objects provide information about states and state transitions to other objects in the system.

- ***REQ Relations.*** The object that defines the controlled variable on its external interface encapsulates the controlled variable function, accuracy, and timing constraints. These objects use the state information (terms and modes) provided by other objects in the system in defining the controlled variable functions.

Heuristics for choosing objects or allocating requirements information among objects are still under investigation. In particular, subsequent work will seek to define more detailed heuristics and address issues of scale-up (hierarchical structures and subsystems).

2.2 THE REQUIREMENTS DATA MODEL

This section describes the information content of a requirements specification developed using CoRE in terms of a document- and standard-independent data model. Since the model is still being developed and the focus is on conveying an intuitive understanding of the approach, only the first few layers of the model are given. Subsequent sections address why particular information should appear in requirements, how it should be developed and used, and how it should be presented.

2.2.1 Notation for Describing the Data Model

The underlying model for the requirements data model was derived from the semantic data model

(SDM) (Hammer and McLeod 1981). SDM represents data as a collection of objects[1]. The SDM groups objects into collections called classes. A given object can simultaneously be a member of one or more classes. If C_1 and C_2 are classes, C_1 is a subclass of C_2 if every object that is a member of C_1 is also a member of C_2.

Associated with each object is a set of attributes, which are named references to related objects in the database. The classes in which an object is a member determine the attributes of that object. Attributes can be single-valued or multi-valued.

Figure 3 shows the graphical notation used to describe the requirements data model. A class is represented by a rectangle. The name of the class appears inside the rectangle. A rectangle contained inside its superclass represents a subclass. Directed arcs to other classes represent the attributes of a class. The arcs may be labeled with the name of the attribute. Arcs that terminate with a single arrow represent single-valued attributes. Arcs that terminate with a double arrow represent multi-valued attributes.

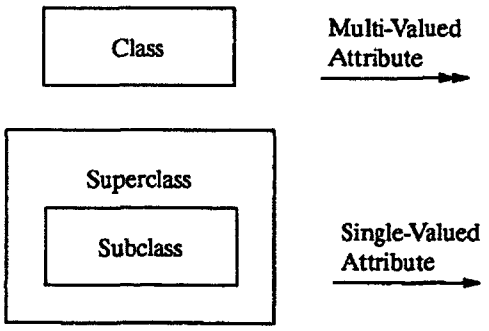

Figure 3. Data Model Notation

2.2.2 The Data Model

Figure 4 illustrates a top-level view of the information that appears in a requirements specification developed using CoRE. The requirements consist of a set of requirements objects[2] (which are modeled using the SDM objects and classes), a set of classes, and a set of relations on the classes.

Associated with each class are a set of attributes. The attributes record common information about objects in the class. For example, a set of similar input or output devices might be modeled as a class. Common characteristics such as the fact that a two-position switch was used is given as part of the class specification. This is illustrated in the warning displays in the FLMS specification which have the use of the operator's CRT screen in common. The class relations include data modeling relations, such as subtype and aggregation, and relations that are particular to a given domain. For example, in the air traffic display domain there may be a relation named watches from the host aircraft class to the potential threat aircraft class.

Associated with each object are an interface and encapsulated information. The interface contains requirements information that engineers specify in the object and that is visible to other objects in the requirements. Monitored variables, attributes whose values the system must be able to determine,

1. The objects and classes in SDM are distinct from the objects and classes that appear in the requirements data model.
2. The issues associated with subobjects or subsystems are not being addressed at this time.

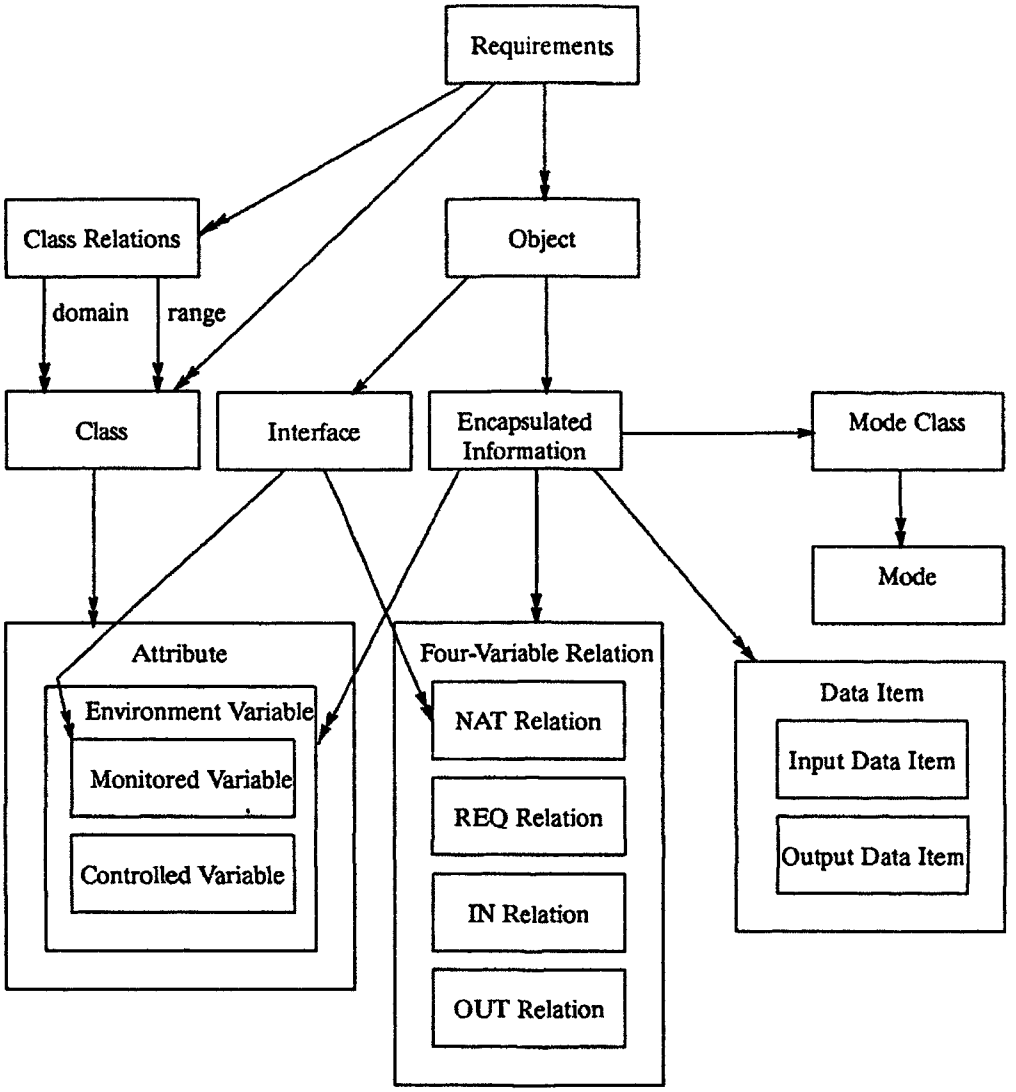

Figure 4. Data Model of Software Requirements

and the NAT relation may appear in the interface. Encapsulated information contains requirements that are not visible to (i.e., may not be used in) other objects. Environment variables, the four-variable relations, data items, and mode class definitions may appear in encapsulated information.

As shown in Figure 4, the class attribute contains the specifications of the environmental variables in the subclass Environment Variable. Environment Variable has subclasses for the monitored variables and controlled variables. Since a number of objects may use one monitored variable, the engineer may define it in an object interface. Controlled variable specifications are private; hence, they are not related to Interface.

The input and output data items are also encapsulated information since the details of device interaction are hidden. Data Item has subclasses Input Data Item and Output Data Item. These contain all the details of device interactions.

Four-Variable Relation has subclasses NAT Relation, REQ Relation, IN Relation, and OUT Relation. Figure 5 shows these subclasses in more detail. The REQ, OUT, and IN relations each consists of a

set of functions. Each of the functions may have an associated tolerance and delay. The NAT relation also consists of a set of functions.

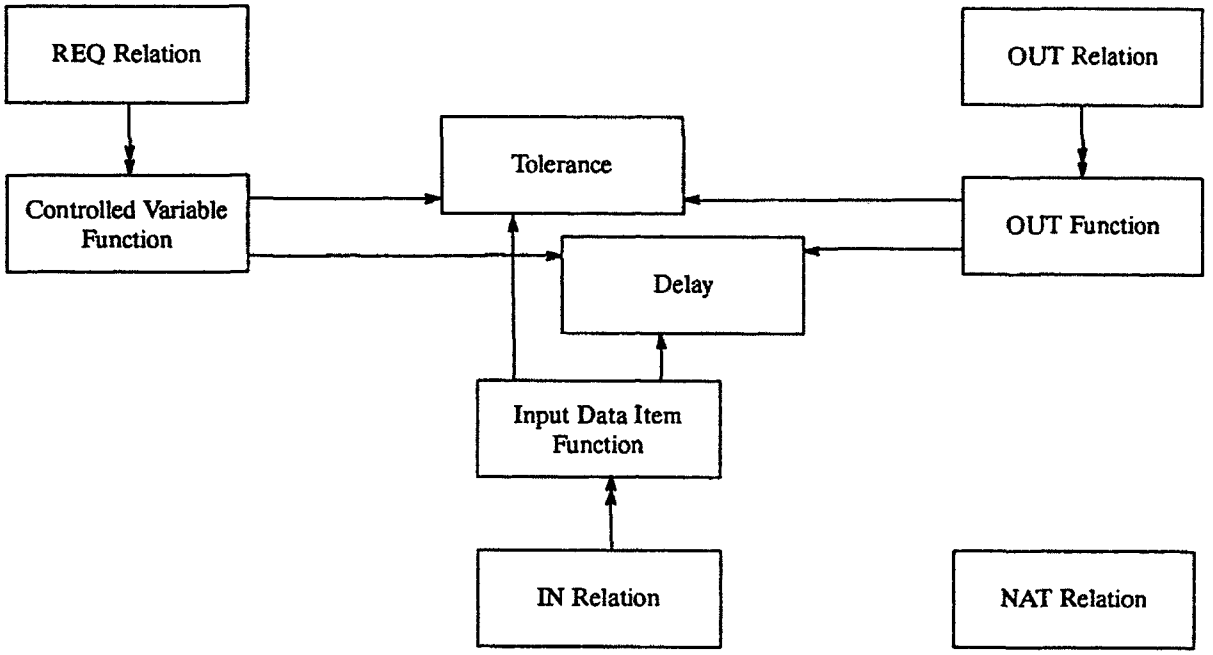

Figure 5. Data Model of the Four-Variable Relation

3. THE CONCEPTUAL METHOD

To arrive at a practicable requirements method, the formal models must be fleshed out with an overall process and a variety of techniques for actually producing work products. This section gives an overview of such a process, then describes the specific methods applying to each of the development steps. The intent is to convey an intuitive understanding of how CoRE would be used in practice. Subsequent reports will provide more formal and detailed description.

3.1 PROCESS OVERVIEW

Since CoRE is still under development, the requirements analysis process (i.e., a description of the method activities with required inputs and expected outputs) supporting it is not fully defined. This will be the subject of a subsequent report. However, the current model and supporting methods imply much about the process and have been refined with an understanding of the basic steps and their products. This section, gives an overview of the envisioned process to convey a sense of how requirements would be developed using CoRE and to provide a framework in which the application of the component technologies can be better understood.

The process described is necessarily idealized in the sense that no real process would (or should) exactly follow the sequence of steps outlined. In practice, different parts of the specification may be resolved to greater or less detail at a given time, depending on which parts of the system are better understood, which are most stable, and which represent greater risk. The constraints of a sequential presentation prohibit the sort of recursion and iteration typical in a real specification process. Instead, the usual convention of presenting the process and its products essentially top-down is followed since this gives the clearest exposition.

Table 1 outlines the key steps of the process. With each process step, the table identifies the key product in terms of the document-independent data model. Finally, it identifies one or more possible representation techniques as used in the discussion examples and FLMS specification.

Table 1. The Requirements Process

Process Step	Product	Applicable Representation
1. Define the object model.	Structure of environmental objects and relations	Information model diagram Object definitions
2. Identify monitored and controlled variables.	Environmental object attribute definitions	Context diagram Attribute definitions

Table 1, continued

Process Step	Product	Applicable Representation
3. Specify environmental constraints.	NAT relation definitions	Tables Equations
4. Specify externally visible behavior.	Controlled variable functions and tolerances (REQ)	Tables Equations
	System mode class definitions	State transition diagrams Event definitions
5. Specify hardware interfaces.	Input data item specifications (IN relations)	Input data item template Tables
	Outputs data item specifications (OUT relations)	Output data item template Tables

How much of these steps the software analyst must do as part of the software specification process depends on what the system analyst did in the systems analysis step and what input he provides the software analyst. In theory, the systems analyst should do steps 1 through 4 for the system as a whole during systems analysis. Systems analysis should determine and specify what aspects of the environment the system must monitor and control, what the system as a whole must do to the controlled variables, and how the environment constrains the required behavior. Where the system analyst allocates the job of monitoring and controlling certain variables to the software, these parts of the system specification then become part of the software specification. The software specification provides additional constraints imposed on the software by the system hardware.

Since, in practice, these steps are incompletely carried out or carried out using a different approach, this discussion (and CoRE) assumes nothing about the specific processes or products of the systems engineering step except that they contain certain basic information. CoRE assumes only that the software analyst can determine from the system specification the jobs that the software is supposed to do, the events in the world to which the software is expected to respond, and the things the software controls to perform its tasks. The specification may be in any form, including English prose.

The overall approach begins by identifying and formalizing the aspects of the environment of interest in the form of a set of interacting objects. The object model captures the relations among groups of environmental quantities of interest, such as linked requirements that are likely to change together, and provides a document-independent framework for organizing the software requirements. The engineer completes the behavioral specification by specifying that behavior in terms of the standardized model for embedded systems.

3.1.1 Step 1: Define the Object Model

The first step in the process models the system in its environmental context as a set of objects, attributes, and interobject relations. The object model establishes the top-level architecture of the

system requirements data model. The engineer uses it to organize information about the system environment by allocating the environmental quantities of interest to the attributes of objects in the model. Grouping the attributes into objects helps abstract from irrelevant detail, provides an organizational mechanism for managing change, and provides a basis for capturing relationships between groups of environmental attributes.

The engineer chooses the objects to capture and organize distinct concerns relative to the software. For example, objects capture devices such as pumps, pressure vessels, or displays that the system monitors or controls as objects. The object organization helps separate concerns by grouping environmental attributes that the engineer should consider together. The object organization also models aspects of the environment that may change together, such as the fact that there may be multiple targets and that the attributes of each target are related. Often, objects will correspond to physical objects in the environment but this need not always be the case.

The product of the object modeling is an initial specification of the object structure. Each object includes in its specification its name and the names of each environmental variable associated with the object. Where there are relations between system objects, the object model also defines these and specifies the related objects and the name of the relation. It also gives any constraints on the cardinality of the relation. For instance, if the system must track up to ten separate targets or there must be exactly one shut-down switch for each pump, the specification provides these characteristics for the relations.

The engineer can graphically model and use the object organization to understand the system and its context. Since the organization into objects provides a basis for separation of concerns, the object organization carries through the specification of detailed requirements. The engineer encapsulates details of requirements that are likely to change or are unnecessary to understand in the specifications of particular objects. This helps limit the information that the engineer must read to answer a specific question about the requirements. Similarly, he can constrain requirements changes to a small number of objects for a broad class of possible changes.

3.1.2 Step 2: Identify Monitored and Controlled Variables

The engineer captures the detailed, behavioral requirements in terms of the environmental variables of interest. In this step, he identifies the physical variables monitored or controlled by the system (if not provided by the system specification), and explicitly defines, and represents them by mathematical variables. Environmental variables include physical quantities such as temperature, pressure, or altitude. They also include information characterizing aspects of the environment of interest like the number and types of a set of target aircraft. He models each such distinct quantity (or set of quantities) of interest with mathematical variables.

The engineer defines each variable by giving its name, its type, its physical interpretation, and its required precision. He can use graphic diagrams (e.g., data flow diagrams) to describe the dynamic interaction of the system with its environment in terms of monitored and controlled variables. He typically gives physical interpretations of the variables in prose or picture format.

3.1.3 Step 3: Define Environmental Constraints

Once the engineer defines monitored and controlled variables, he must specify the environmental constraints between them. Both the natural world and other man-made systems in the environment,

including the system's own hardware devices, constrain the possible values of environmental values. For example, the maximum rate of climb of an aircraft, maximum rate of change of velocity of a target, the refresh rate of a display, or the maximum slew rate for an antenna are all externally determined constraints on variables that the software helps to monitor or control. The engineer captures these in the model in terms of the mathematical relation called NAT. The NAT relation determines what environmental quantities the developer will and will not be required to account for in the system behavior. This is important, both in determining the completeness of the specification and in assessing the feasibility of requirements. For instance, the environmental constraint that an aircraft has a maximum and minimum possible altitude and a maximum possible rate of change tells the developer what kinds of values he must account for in defining, say, the information represented on the head-up display, and limits how frequently the system must sample the altitude to provide a given accuracy.

3.1.4 Step 4: Define Externally Visible Behavior

The heart of the specification is in the definition of the software's externally visible behavior. The groundwork done in carefully specifying **what** aspects of the environment are of interest to the software in terms of monitored and controlled variables allows a systematic approach to the specification of required behavior (**how** the software interacts with the environment).

The mathematical relation between the monitored and controlled variables called REQ captures the relationship among environmental variables that the software must maintain. The method specifies required behavior by defining the values of the controlled variables (i.e., the visible system outputs) at all times. For every value controlled by the system such as the value of a display, the state of a valve, or the attitude of a control surface, the specification provides a function specifying the value of that variable for all possible states of the system. The output function captures the idealized, required behavior (i.e., the displayed value equals the altitude). An additional expression (i.e., tolerance of two feet) allowed deviation or tolerance. The engineer writes the output functions entirely in terms of monitored variables, ensuring that the specification describes only externally visible behavior. Finite state machines (called model classes) capture the history of events, where the required outputs depend on the history of events.

Since mathematical functions of the monitored variable states define the values of outputs, there are well-defined notions of completeness and consistency that the engineer can apply to the specification. For instance, the external behavior specification is complete when every controlled variable has a complete function; i.e., the specification assigns one of the possible values of the controlled variable to every possible system state. It is consistent if no state leads to two distinct outputs.

3.1.5 Step 5: Define Hardware Interfaces

To develop a design, the engineer must also know what resources are available to determine the values of the monitored variables and set the values of the controlled variables. These resources are the actual inputs and outputs provided by the system hardware or, if this information is not yet available, a suitable abstraction of the expected inputs and outputs. On the input side, this requires identifying each distinct system input or an abstract data item representing the input. The engineer creates an input data item by giving a name and type to the input and describing its characteristics. He can do this by filling in a standard template for describing data items. On the output side, he applies a similar process to specify output data items.

The final steps in the process establish the relations between the monitored variables and the system inputs and the controlled variables and the system outputs. To determine that an implementation is feasible, it is necessary to know if the system can determine the external characteristics it must monitor the required precision from the inputs actually or typically provided. The relation called IN, from monitored variables to inputs, specifies time. Similarly, the relation called OUT defines the relation between the system outputs and the controlled variables. The IN and OUT relations capture the correspondence between actual or expected hardware behavior and the required system behavior relative to the environment.

While this approach is successful for the small examples used so far, this is one area where much work remains to be done before the method can scale up. Establishing the IN and OUT relations is not difficult when the inputs and outputs are known, and there is a simple relation between these values and the environmental variables. However, the work to date has not addressed more complex relations such as instances where several inputs are required to determine the value of a monitored variable, or the hardware is expected to change dramatically in form or function. The Consortium is currently investigating these issues.

3.2 DEFINING OBJECTS

The engineer can describe the externally visible behavior of a system solely in terms of monitored and controlled variables. However, treating each of these variables as an independent component of the system's environment will result in the loss of important requirements-related information. For example, two monitored variables, such as the pressure and temperature of the liquid in a vessel, may be interdependent because of the nature of the environment. System requirements may also impose dependencies, for example, if a system interacts with multiple devices of given types, there may be related groups of variables based on which monitored values are associated with which controlled values. Finally, the potential for variation in requirements may introduce dependencies, for example, if the devices that monitor two distinct variables will always be replaced as a unit. The engineer can treat these various dependencies in a uniform manner by clustering the monitored and controlled variables and representing them as **attributes of related object classes** within the requirements model.

The engineer may define the association between a class of objects and the set of variables which are its attributes in either a bottom-up or a top-down manner. A bottom-up approach identifies a monitored or controlled variable (for example, a monitored liquid level), and then identifies the object with which the variable is associated (for example, the liquid in a vessel). A top-down approach identifies a class of objects (for example, targets for a weapons system), and then identifies the variables that characterize these objects (for example, position and velocity vectors).

In general, the engineer can define **object classes** as components of the "problem space," or, more specifically, as components of the system's environment. However, this does not mean that he can choose a useful set of object classes by a naive enumeration of a system's surroundings. The choice of objects requires an interaction between the model builder's understanding of the system's purpose and the characteristics of the system's environment. As an example, consider the set of tracks detected by a radar system. A civil air traffic control system may consider all the tracks to be "flight" objects with very similar significance. However, a military command and control system may divide the tracks between "friend" objects and "foe" objects of very different significance. Similarly, a system may consider two mechanically connected devices as two independent objects if their behavior is unrelated, while it may consider two physically separated devices with related behavior as parts of the same object.

Object types need not be tangible components of the physical world. For example, a mode class can represent the system state transitions (also sometimes called a "procedure" or "plan") coordinating two or more objects formally as a **mode class**. A mode class is associated with an object that encapsulates the details of how the current state (**mode**) is determined, providing summary state information as needed to specify other objects in the requirements. Such an object need have no (in fact, usually will not have) physically independent existence within the problem domain.

For example, the engineer might define the control of a burglar alarm using common modes such as unarmed, armed, and alarm sounding. These modes would be the visible properties of an alarm state object which provides the state information used to define requirements in objects associated with the alarm bell, intrusion detectors, and system users. However, the alarm state object corresponds to no physical object in the system or environment.

Simple systems may have only one such state or mode object. However, more complex systems may contain many such objects as well as nontangible objects of other kinds. In this method, it is the abstraction and encapsulation properties that motivate the use of objects, so the analyst is free to create a requirements object wherever he can encapsulate requirements details. While physical components in the environment may provide a useful starting point for choosing appropriate requirements objects, they can be no more than a useful heuristic. Ultimately, the best choice of objects depends on what qualities in the environment are likely to change together (or separately), and what requirements are likely to change over time. Encapsulating information to reduce complexity and confine change results in specifications that best meet the goal for ease of change. For embedded systems, those objects also typically coincide with physical components since these tend to change as a unit.

One type of object that is useful is the **interface object**. These objects, as their name implies, are an interface between the system and an object in its problem space. They encapsulate details of the interface (hardware devices and the nuances of objects in the domain, for instance) that are particularly likely to change.

Assigning each attribute to a class of objects generally leaves some dependencies among the attributes unexpressed. The requirements model must therefore define relations between object classes. In general, relationships are based on some association between objects of one class and objects of another. For example, in a weapons system, the relationship "is assigned to" exists between weapon objects and target objects. A relationship serves as an organizing framework for information about the association between the related classes. For example, relationships may connect one or many objects (more than one weapon may be assigned to a single target). Also, a relationship may be mandatory or optional (a target may have no weapon assigned to it).

A variety of notations can express object classes, relationships, and attributes. The diagram illustrated in Figure 8 on page 44 is an example of one such notation, the entity-relationship diagram provided by many CASE products.

3.3 IDENTIFYING MONITORED AND CONTROLLED VARIABLES

The ability to identity and define the monitored and controlled variables depends on having an adequate description of the system boundary and the information crossing it. This understanding is also essential to the specification of NAT and REQ—exactly what information belongs in which relation depends on where the engineer draws the boundary.

Since drawing these boundaries (equivalently, allocating tasks to system components) is part of the system development process, the Consortium's current work does not deal with it explicitly. However, to illustrate the kinds of information that must be available to determine the environmental variables, as well as the contents of REQ and NAT, the results of the system development process must be considered. While CoRE makes no assumption about the system design process or the form of its products, it is obvious that it will be easier to use the results of a compatible process (e.g., one that is also based on modeling the system as objects) than one based on a completely different paradigm. Since it gives the clearest illustration of the issues, the process of determining the environmental variables and relations based on such a model, those developed as part of the CASE/Real-Time Method, is discussed. Others commonly available models also contain the same kinds of information (e.g., Shlaer/Mellor).

For CoRE, it is only important that the information be available. This illustration only serves to demonstrate that the graphic and other methods can make such system decisions quite clear.

3.3.1 Defining the Domain and System Boundary

For the sake of this illustration the initial step in developing the system requirements is assumed to model all of the relevant aspects of the problem, including those implemented by hardware, humans, natural process, and so on using the CASE/Real-time Method. The collection consisting of all aspects of the environment, system, etc., that are relevant to the problem is called the domain, and a model of information in the domain a domain model. The domain model gives context for the system. It describes the environment in which the system operates: the objects with which it interacts, the attributes of those objects that the system may monitor and control, and the laws that govern the behavior of those attributes.

The domain model is a set of formal models that describe a problem in its entirety without yet representing decisions about which parts of the problem will be automated. One component of the domain model representation is the domain information model. This is a data view of the pieces in the domain represented as objects. An entity-relationship diagram represents this information. The second is the domain transformation model. This view represents the dynamic relationship between elements of the problem domain in the form of communication objects. It shows the information flow, sources, sinks, and processes (state machines) using a real-time data flow notation (e.g. Ward/Mellor or Hatley/Pirbhai).

System design embodies the process of deciding which jobs will be automated and by what part of the system. A boundary placed around some set of the objects in the transformation model represents such decisions in the object mode. This object model characterizes each object can be characterized as outside the system, inside the system, or on the interface between the system and its environment. Once the engineer decides which objects are in the system and which are not, he can readily identify the monitored and controlled variables as well as which relations in the model are part of REQ and which are part of NAT. In particular:

- The monitored and controlled variables correspond to data from objects in the environment that interact with objects on the interface. That is, the data that crosses the system boundary going in defines what the system monitors, and the data going out defines what the system controls.

- The objects and relations inside the boundary represent the behavior that the engineer must implement as part of the system. These become part of REQ.

- The objects outside the boundary that represent sources or sinks for the environmental variables represent relations that exist in the problem domain but are outside the system itself. These ultimately define part of the NAT relation.

For example, Figure 6 illustrates a domain transformation model. It shows the objects identified in the information model for the FLMS, the boundary between the system and the domain, the information that the objects exchange, and the direction of information flow. The figure identifies three types of objects: those outside the system and hence part of the domain (Tank, Watchdog, Pump and operator), those wholly within the system (InOperation), and those that are interfaces between the system and its domain (Operator Interface, Watchdog Interface, Fuel In Tank Interface, and Pump Interface). Figure 6 represents the system boundary by drawing a dashed line through all the interface objects, enclosing the object wholly within the system (here, just InOperation). The following sections discuss these issues in more detail.

3.3.2 Specifying monitored and Controlled Variables

When the engineer determines the system boundary, he must decide which environmental quantities the system must monitor and control. In any system complex enough to be interesting, there will likely be more than one choice about what quantities to call monitored variables. Even for a system as simple as the FLMS, there could be a debate about whether the monitored variable for the fuel should be its volume, its level at some particular (but which?) point in the tank, the pressure in the tank, and so on. In many cases, the engineer will be free to choose which of a set of related variables as monitored and which to compute from the others. Procedures and heuristics for choosing a good set of monitored variables remain an area of active investigation and will be discussed in subsequent reports. This demonstration uses the simple heuristics of first choosing those quantities that seem to result in the simplest specification, and then choosing variables that are not redundant (no one can be calculated from any combination of the others). This approach appears sufficient for the small examples.

The simplest specifications result if the engineer chooses the controlled variables to reflect the physical devices most directly affected by the computer (e.g., the heater rather than the external temperature) since the state of the controlled variable is directly dependent on the outputs of the system. In some cases, there may be a choice about whether the engineer should treat a variable as monitored, controlled, or both. For instance, the FLMS monitors fuel level and controls the pump shutdown switch. Given that opening the switch means the fuel level cannot change, it is possible to argue that the system also controls fuel level. This relationship of indirect control, however, is a property of the domain that results from a hardware design decision. It is better to state that the system directly controls the pump shutdown switch. If it is necessary to interpreting the requirements, the NAT relation can capture the relationship between the switch and the fuel level (see Section 3.4).

The engineer should choose variables and variable names that are commonly used for technical communication concerning such systems. For instance, an embedded system for an attack aircraft might have variables representing airspeed, angle of attack, altitude, and so on. This makes the specification more understandable to customers and systems engineers. These guidelines will become more specific, and may change, as experience in applying the method grows.

Once the engineer chooses the environmental quantities, he represents each with a mathematical variable. The specification of the variable must include its name and its type. It must also describe precisely the correspondence between the variable and the environmental quantity the variable represents. This may

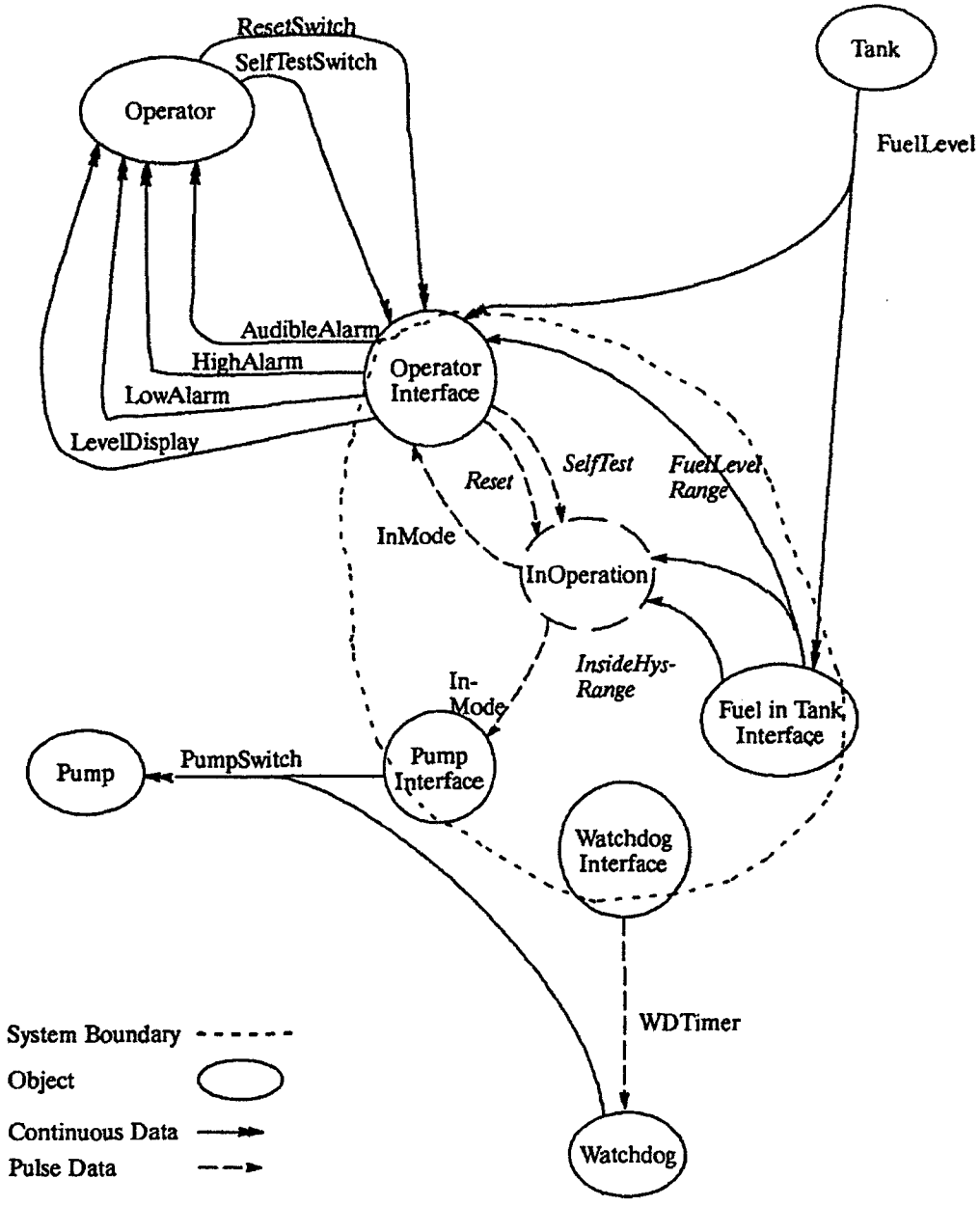

Figure 6. Fuel-Level Monitoring Domain Transformation Diagram

require the use of diagrams or pictures where such variables represent relationships between environmental entities (e.g., angle of attack).

Typically, there are constraints on the possible values a variable can take due to properties of the environment. For instance, a tank will have a maximum volume and fill rate, and an aircraft will have a maximum velocity, altitude, or rate of climb. The NAT relations capture these constraints.

3.4 SPECIFYING ENVIRONMENTAL CONSTRAINTS: DEFINING THE NAT RELATION

The NAT relation specifies behavior that is possible in the problem domain. These include both constraints on the individual variables in the domain (such as the maximum volume of a tank) as well

as constraints imposed by the physical world on possible relations among the environmental quantities of interest. For instance, the maximum rate the fuel level can change with the pump on is a property of the environment in which the software must operate.

Describing the NAT relation constrains the set of possible requirements that must be specified in the REQ relation. For example, if the aircraft has a maximum rate of climb of 30 feet/second, it is not necessary to specify required behavior such as values of a rate-of-climb display for values above 30 feet/second. In general, any combination of values that are not possible in NAT need not be specified in REQ (though the wise developer will account for likely changes).

The engineer may define NAT by starting from a domain transformation model, such as the one for the FLMS shown in Figure 6. To define NAT, the engineer looks for relationships between attributes of objects in the domain (that is, between monitored and controlled variables). Any formula that governs relationships between values of environment variables is a candidate for inclusion in NAT. He does not need to consider the variables shown inside the boundary. The software introduces these quantities, and they are not parts of the domain. The REQ relation specifies these.

Defining NAT is a task for experts in the domain, since they have the requisite knowledge of laws of interaction. NAT is expressed using notations appropriate to the domain. The FLMS, for instance, uses a differential equation to capture the maximum amount that the fuel level can change over a given period of time. An event table also describes the relationships between two components of discrete domains (the reset switch and the pump switch).

Since environmental constraints may exist on individual variables as well as relationships among variables, an expression defining NAT may involve any combination of monitored or controlled variables. Some will involve only one monitored or controlled variable. For example, the environmental constraints on maximum and minimum fuel level due to the size of the tank and method of measurement need only the monitored variable FuelLevel. Some constraints will involve both monitored and controlled variables (or multiples of either kind). In the FLMS, setting the pump switch to Off implies the change in FuelLevel will be zero until the system is reset. This can be stated as the implication:

$$\textbf{if } \text{PumpSwitch(t)} = \text{off } \textbf{then} \quad \left| \frac{d(\text{FuelLevel(t)})}{dt} \right| = 0$$

This constrains the possible combinations of variable values by showing that, once it shuts off the pumps, the system need take no further monitoring action. The fuel level will remain in its current state.

3.5 DEFINING VISIBLE BEHAVIOR

The REQ relation specifies what the software must do—its externally visible behavior in terms of monitored and controlled variables—is described with the REQ relation. That is, it describes the permitted values of the controlled variables in terms of the possible states of the monitored variables over time. Thus, CoRE needs techniques for capturing changes in state over time and relating such state information to expected behavior.

3.5.1 Specifying the REQ Relation

REQ specifies both the ideal behavior of the system and permissible deviations from that ideal. Ideal behavior is possible only with devices capable of both infinite accuracy and instantaneous reaction

to external stimuli. The hardware devices that the software reads and writes are capable of neither. They are of limited accuracy and take some small but finite amount of time to react to physical properties (in the case of input data items) or to influence them (in the case of output data items). Therefore REQ includes a description of acceptable delays in reacting to changes in monitored variables and a statement of how much deviation can be tolerated from the values computed for controlled variables if the hardware had infinite precision.

To write REQ, the engineer must determine how he wants the controlled variables to behave—that is, what values they are to assume. In general, he determines the values of controlled variables by the values of monitored variables. He must therefore determine which monitored variables influence which controlled variables. Then, for each controlled variable, he defines a function that specifies the variable's value at any point in time. (A controlled variable has exactly one such function. However, the engineer may describe all controlled variables whose values always change together by a single function.)

In most systems, past as well as current values of monitored variables determine the values of the controlled variables. The value of current conditions alone is not enough to determine the required behavior. For instance, the interpretation of a button on a digital watch depends on the current mode. The current mode is a function of the number of times the mode button has been pushed. The engineer specifies this type of behavior using finite state machines called mode classes to summarize the relevant history of the monitored variables (see Section 3.5.2). Mode classes capture and encapsulate state information that may be common to many of the controlled variable functions. The use of mode classes simplifies the functions since the mode class objects hide the details of state determination. As the engineer writes REQ, he needs to understand how the mode classes, and the modes within each class, characterize system behavior.

The engineer specifies the required values of the controlled variables in the form of piecewise-continuous functions reflecting the discrete nature of digital systems. He expresses the domain of these functions in terms of conditions on the monitored variables, or on the current mode, or both. He therefore partitions the domains of the functions that describe the REQ relation into formulas based on system modes, and events and conditions defined on environment variables as necessary. It is generally easiest to write the function to define ideal behavior, adding required timing constraints and tolerance as the details of the requirements become better understood (see Sections 3.8 and 3.9). If he can establish upper bounds early in the process, these can serve as place holders until more of the system details are decided (up to, and including, the selection of hardware).

Some controlled variables need initial values. Often these do not fit the paradigm used to describe the behavior at other times—specifically, it is not a function of the same monitored variables. For example, in the FLMS, the HighAlarm is initially on. At all other times, it is a function of FuelLevel. In cases such as these, it is convenient to specify the initial value separately.

The following example, using the FLMS, illustrates how REQ might be written. The focus is on the portion related to displaying the fuel level to an operator. There are three environment variables of interest:

- LevelDisplay, a controlled variable of type length. In the implementation, it is the value conveyed on a CRT screen, labeled FUEL LEVEL.

- FuelLevel, a monitored variable also of type length. Its value is the level of the fuel in the tank, in centimeters, measured in a specified way.

- Time, measured in seconds. The variable Time is the time elapsed with respect to a fixed, but arbitrary, reference point *t*.

In this example, the value of LevelDisplay depends on the system's execution history. Normally, it is the value of FuelLevel rounded to one decimal digit. However, if the operator pushes a "self-test" button, then the value of LevelDisplay is different—it is a function of the amount of time since the button was pushed. After 14 seconds, its value becomes that of FuelLevel again. Thus, the function giving value of LevelDisplay is not just a function of current system conditions. A mode class needs to capture which event occurred most recently: the operator pressing the self-test button, or 14 seconds elapsing since the last press. The InOperation mode class of the FLMS has modes: Operating, Shutdown, Standby, and Test. If the operator has pressed the self-test button within the last 14 seconds, the system will be in Test mode; otherwise it will be in one of the other three.

Define the function:

$$\text{Round}(x, y) = z$$

in which real z equals real x rounded to positive integer y decimal places. The value of the controlled variable LevelDisplay is then given by the following set of equations, where t_0 is the instant at which the system entered Test mode:

$$\text{LevelDisplay} = \begin{cases} \text{Round(FuelLevel, 1)} & \text{if in Operating, Shutdown or Standby mode} \\ 0.0 & \text{if in Test mode and } 0 \le \text{Time} - t_0 < 4 \\ \lfloor(\text{Time} - t_0\rfloor \times 11.1 & \text{if in Test mode and } 4 \le \text{Time} - t_0 < 14 \\ 0.0 & \text{if in Test mode and } 14 \le \text{Time} - t_0 \end{cases}$$

If the system is not in Test mode, the value of LevelDisplay depends on the monitored variable FuelLevel. If the system is in Test mode, then the value of LevelDisplay depends upon the monitored variable Time and how long the system has been in the mode. This function is a good example of the typically piecewise-continuous behavior of embedded systems in that the controlled variable value shows points of discontinuity on entrance to and exit from Test mode.

This equation defines a function controlling LevelDisplay that is part of REQ. This example specifies the function by partitioning its domain by the system mode and, within the system mode Test, by conditions defined on the environment variable Time.

This specifies the ideal behavior. In normal operation, the level display is simply the fuel level, rounded to one decimal point. In Test mode, the level display is a function of the length of time that the system has been in that mode. In reality, complete precision is not possible. The engineer must specify an acceptable deviation between the value of FuelLevel and the value of LevelDisplay. He must also give an acceptable delay between the time that FuelLevel changes value and the time that LevelDisplay reflects that change (see Section 3.9). For instance, in modes other than Test, a deviation of 0.5 centimeters from FuelLevel, and a delay of up to 500 milliseconds from the time FuelLevel changes to the time LevelDisplay changes, are acceptable in the implementation. Test mode allows a clock accuracy of plus or minus 1 millisecond. It permits a delay of up to 500 milliseconds between the moment the system recognizes a change in time and the moment LevelDisplay shows the new value.

3.5.2 Describing State and State Transitions

All requirements methods for real-time systems must provide methods for representing the states and state changes that a system must track and to which it must respond. Since there is interest in

describing the system behavior in terms of well-defined mathematical functions of the system state, the method for capturing state information must result in specifications that are also mathematically well-defined. Thus, both the Consortium's underlying model and the methods used are more rigorous than is common. This places some burden on the analyst in forcing him to be precise in his meaning; however, the effort is repaid in specifications that are both precise and amenable to rigorous analysis for completeness and consistency.

Two goals in developing CoRE are first, that it be usable with little formal training beyond that typical of member company line engineers and second, that it be usable without deep understanding of the underlying formalisms. For these reasons, this report does not treat the underlying mathematical model in detail. There is a small, preliminary discussion of the underlying model, with most of the discussion focusing on conveying an intuitive understanding of CoRE. It should be clear from the discussion that analysts can develop rigorous specifications with the intellectual tools they now have.

In general, embedded systems implement two distinct kinds of behavior. In some cases, the outputs are a function of the present inputs only. For example, whenever a reset button is pressed while all operating conditions are within an acceptable range, the system must turn on the pump. In other cases, the outputs are functions of both the current inputs and the time-ordered sequence of previous inputs. For example, one mouse click followed by another a second later may have a different meaning from two in quick succession. This is analogous to the difference between combinational and sequential logic in hardware circuits. The remainder of this section discusses how the engineer captures these distinct kinds of state information. Current state information is expressed in terms of conditions and events, and sequential information in terms of modes.

The goal is to characterize the interesting aspects of the environments state in a rigorous and systematic way: systematically, because this provides the developer the most guidance in understanding what question to ask next and when the job is done; rigorously, because the meaning must be precise and specifications must be produced that can be meaningfully analyzed in a mechanical fashion.

3.5.2.1 State Conditions

At the most primitive level, conditions written in terms of monitored variables capture the system state using conditions written in terms of the monitored variables. A condition is a predicate that characterizes the state of the system for some measurable period of time. For instance, *altitude* > 500 ft. and *FuelLevel* > 30 cm. might be conditions, the first representing all those states of the aircraft where its altitude is above five hundred feet and the second characterizing all states of the FLMS where the fuel level in the tank is above 30 centimeters.

Formally, by defining conditions, the engineer is defining sets. The universe from which he draws the sets is the superset of all possible values of the monitored variables (i.e., all possible values of $\{m'_1, m'_2, \ldots, m'_n\}$). A particular state of the environment (e.g., where the fuel level is higher than 30 centimeters) is equivalent to the set of monitored variable values where the variable FuelLevel has a value greater than 30 centimeters. The approach makes sense because the engineer chooses the monitored variables so they represent all the quantities of interest in the environment. This allows him to characterize aspects of the environment of interest (i.e., states) in terms of the values of these variables.

Formally, a condition is a truth-valued function that characterizes a distinct set of system states. The function's domain is all possible assertions about the values of monitored variables; the range of the

function is the values true and false. A truth-valued function of this sort (i.e., one that only takes on the values true or false), is called a predicate. A predicate characterizes a set by giving a mapping from elements of the domain to the value true for those elements in the set being characterized. This is the usual way of writing sets in algebra or in programming, so it is generally well understood by systems and software engineers. For instance, the set of integers greater than ten would be written $\{x \mid x > 10 \text{ and } x \in I\}$. Similarly, anyone who has programmed has become used to characterizing the set of program states that select the true branch of an if statement by writing a predicate on the states of the program's variables in the form of a boolean condition (e.g., `if (x>10) then...`).

The developer can use the same technique to characterize state information in requirements with the convention that he use only monitored variables in the expressions and (the usual convention) that the expressions formed evaluate to true or false based on the values of the quantities measured by the variables in the real world. This convention is typical of real-time methods (e.g., Ward/Mellor or Boeing/Hatley), adding primarily the constraint that the variables used in state expressions be drawn from the set of monitored variables.

The engineer can describe more complex conditions in the usual way by forming boolean expressions over simple conditions. For instance:

(FuelLevel > 5 cm. AND FuelLevel < 30 cm.) OR SelfTest = Pressed

3.5.2.2 Events

For real-time applications, the interest is not only in the current state but in those points in time associated with state changes (e.g., the moment a button is pushed or the fuel level has been too high for some threshold interval). The engineer can capture these moments as changes in the values of conditions; such a change is called an event. Whereas conditions persist for measurable periods of time, events occur at single points in time; i.e., in the idealized model, events are instantaneous.

Events are a relation between the state before and after the change. For example, the event associated with FuelLevel < 30 cm. refers to any state change where the value of the monitored variable FuelLevel was thirty centimeters or greater and became less than thirty centimeters. An event is formed from a condition using the following notation:

@T(condition)

This describes any moment at which there is a state change from a state in which the condition is not true to one in which it is. Thus, the event given above is written as *@T(FuelLevel* < 30 cm.*)*. Similarly, *@F(condition)* denotes any moment condition becomes false.

Often, the engineer needs more information about the state to describe an event than just what conditions have changed. The WHEN clause describes an event in which one condition changes at a time when another holds:

@T(condition1) WHEN condition2

For instance, *@T(SelfTest = Pressed) WHEN FuelLevel* < 30 cm refers to the event occurring when the self-test button is pressed at a time the fuel level is below thirty centimeters. The event occurs only if the value of *condition1* changes. The statement *@T(FuelLevel < 30 cm) WHEN SelfTest = Pressed*

refers to the event of the fuel level going below thirty centimeters at a time the self-test button was being held down.

3.5.2.3 Modes

Where required behavior depends on the order of events in time, the engineer captures this using mode classes and modes. That is, whereas he captures combinational behavior in the method using conditions and events, he captures sequential behavior using modes.

Formally, a mode class is a finite state machine defined on the states of the system-monitored variables. Each state of the machine, called a mode, corresponds to a set of system states. Since events describe changes in the system state, the machine "input" is events. Finally, the set of modes in a class must partition the set of possible states. Thus, the system is always in one and only one mode of a given class.

Any large specification has a variety of functions concerned with very different aspects of the system state. For instance, the flight control system for an attack aircraft must handle navigation, weapons delivery, aircraft integrity, flight control, and so on. The navigation functions depend on factors such as the aircraft latitude, destination, and which navigation devices are being used. The weapons delivery functions depend on the weapons selected, characteristics of the target, flight characteristics of the weapon, and so on. In such cases, where different functions require different state information, it makes sense to have more than one mode class. In general, the engineer should introduce additional mode classes wherever two functions are relatively independent and a simpler specification results. He writes the complete specification as a set of concurrent state machines enabling the output functions.

In the model, a mode class is a named object consisting of:

- A set of modes. The modes of the class partition the states of the system.
- A set of events defined over the set of monitored variables.
- A transition function (next state function) that defines the next mode for each possible combination of current mode and event.
- The initial mode—the mode in which the system starts when initialized.

For example, the FLMS has four distinct states. It must shut down the system only after the fuel level has been out of the safe range for a specified length of time (the ShutdownLockTime). Thus, there is a normal operating mode of behavior (called Operating) and a second mode that the system is in while the fuel level is out of bounds but the ShutdownLockTime has not yet expired (called Shutdown). As long as the fuel level keeps moving between the safe and unsafe levels without staying unsafe for the lock time, the system will move between Operating and Standby modes, sounding alarms but never actually shutting the pumps down. The required behavior changes if the fuel level stays unsafe longer than the ShutdownLockTime. Now, the system must mechanically disable the pumps; they cannot be started again without operator intervention. This distinct behavior is given another mode (called *Standby*). Finally, there is a distinct set of behaviors associated with the self-test button being pressed, so there must be another mode called *Test*.

The engineer makes the specification simpler using mode information than it would be just using conditions. For instance, he can specify the possible transitions between the states represented by

Operating and Standby modes in terms of mode transitions rather than the history of conditions (i.e., fuel level is out of bounds, and the fuel level has been out of bounds continuously, and it has not been out of bounds for more than the ShutdownLockTimes, ...). This makes the output functions easier to specify while retaining formality. Further, he has now largely captured the state information in the mode specifications where it can be read and understood distinct from issues of how it is used.

In addition, since the modes are well-defined state machines, the engineers can do certain completeness and consistency checks on the output functions. Since a mode class partitions the set of possible states, a function will be complete and consistent if it assigns exactly one required behavior to each and every mode in the class. Section 3.10 discusses this in detail.

3.6 SPECIFYING THE IN RELATION

Once the engineer specifies REQ and once the hardware designers select the input and output devices, the engineer can write the IN relation. This describes relationships between monitored variables and hardware-generated inputs that the software reads. Alternately, he may opt to describe the hardware registers as virtual devices. This practice is desirable in systems where the hardware is likely to change, or where the engineer has not fully specified the hardware yet. The requirements will then be isolated from nuances of the hardware.

The engineer starts specifying IN by considering the available set of devices (virtual or real). He associates each device with a set of monitored variables. The FLMS, for example, has a panel device with two buttons on it that correspond to the reset and self-test variables. The task is then to associate these devices with the actual values read (if known). He does this by defining a set of data items. Each data item represents a datum produced by a device. (Some data items represent more than one datum, or are associated with more than one device, but these cases are ignored to simplify the exposition.) For each one, he specifies:

- A unique name.
- The device that produces the datum.
- The actual values as read; for example, the number of bits and the portion of a register receiving a datum (e.g., port C, bits 0 through 3).
- The range of values associated with the data item; e.g., integers in the range [0, 255] or the literal values on and off.
- A mapping of the data item values to the actual input values read.

A data item serves two purposes. First, it states the association between devices and the values they provide. Second, it provides a useful layer of abstraction. The data item represents the value provided and abstracts from hardware-dependent details such as the specific representation conventions, register used, scaling, etc. He therefore expresses the relations that form IN using data items, not the lower level values generated by a device. The domain of the relation is a monitored variable, and the range is a data item.

The engineer can specify IN relation using much the same techniques as for REQ. He can write the relation using piecewise-continuous functions, partitioned by conditions. The main distinction is that

he does not use system modes to specify IN (or OUT). System modes reflect information on system behavior. The mapping to an input device should be independent of such behavior. This reflects the fact that the engineer should define a device's value relative to a monitored variable whether or not a system happens to be reading that device.

The specification of the relation between a data item (or items) and a monitored variable must define the following:

- A conversion equation between the domain of the monitored variable and the representation of the device-generated value.
- The loss of precision from the real world incurred by using the device.
- The delay introduced by the device.

Consider the FLMS example. It includes a monitored variable, FuelLevel. As it happens, there is a simple correlation between fuel level and differential pressure of the fuel tank: the higher the fuel, the greater the pressure. Thus, an input device measuring changes in pressure, called a differential pressure unit, is used to measure fuel level. An input data item called DiffPress is created to show how this device corresponds to FuelLevel. As part of IN, the following relation between FuelLevel and a hardware register is then written:

$$\text{DiffPress} = \begin{cases} \dfrac{\text{FuelLevel} - (-0.01902 \times (B-A) + A)}{1.03803 \times (B-A)} \times 2^8 & \text{if } 0 \le \text{FuelLevel} \le 30 \\ 0 & \text{if FuelLevel} < 0 \\ 255 & \text{if FuelLevel} > 30 \end{cases}$$

where $A \le \text{FuelLevel} \le B$

This converts FuelLevel into an integer value between 0 and 255, which is suitable for representation as an 8-bit, unsigned quantity (which is what the hardware register requires). The values for A and B define the acceptable precision. The delay is specified separately and is based on an understanding of how quickly a differential pressure unit can react to changes in pressure. A value of 0.2 seconds is reasonable.

Other monitored variables correspond to discrete events—the appearance of an aircraft in radar surveillance range, for example—and are better described using events. The system timer in the FLMS is another example of such a device. Although the monitored variable Time is continuous, the corresponding device emits a pulse; this is conveniently related to Time using a data item ClkPulse, wit' the following event-based relation as part of IN:

$$\text{ClkPulse} = @T(\lceil \text{Time} \times 1000 + w \rceil \bmod (54897 + k) = 0\mu s)$$

where $|k| \ll 54897$.

3.7 SPECIFYING THE OUT RELATION

The Consortium's work to date has not dealt with some of the difficult aspects of defining the OUT relation such as the precise specification of user interfaces and specification of the sorts of complex data types typically found in C^{3}I systems. Subsequent work will address these issues. The current work has concentrated on controlled variables where the mechanism of display is trivial or where issues

of the values to be displayed can be separated from the means of display. The use of objects facilitates this strategy since the engineer can hide the details of how he represents a particular value to the user from the rest of the requirements. For instance, the Operator Interface object in the FLMS specification hides the details concerning how the value of the fuel level actually displays on the operator's CRT screen.

The engineer specifies OUT relation using the same techniques that he uses for the IN relation. However, instead of showing how a hardware-provided value maps to a monitored variable, the specification shows how writing a value to a device affects a controlled variable. It shows the effect of actual software outputs on the environment. The specification of an output data item provides:

- A unique name.
- The device that accepts the datum.
- The actual values given to the device (if available); e.g., the number of bits and the register to which the value is written.
- The range of values associated with the output data item; e.g., values "open" and "closed" for a valve.
- The mapping from actual values written to the data item values; e.g., writing 1 boolean results in a valve value of open.

When writing IN, the engineer is typically concerned with truncating a real value into an fixed-length integral representation. When writing OUT, the opposite is true: he often must map a fixed-length representation of a value onto an infinite domain. This may cause difficulties in writing the software, unless he has been careful in writing REQ. The composition of IN and OUT involves a loss of significant digits which cannot be recovered. REQ includes tolerances for this reason.

Another solution is to write REQ so that the values are consistent with the possible precision from existing devices. The FLMS uses this approach for the controlled variable LevelDisplay. The function that specifies its value rounds that value to a single significant digit. In other words, a variable of infinite precision is monitored but one of fixed length is controlled. The engineer matches precision of LevelDisplay to that of the differential pressure input device, so that he avoids the loss of precision. He can write OUT without describing it.

3.8 SPECIFYING TIMING CONSTRAINTS

Timing constraints describe the way variables behave relative to each other with respect to time. The engineer describes the constraints by a tolerance value representing a maximum permissible delay between the change in value of one variable and the corresponding change in value (to be implemented by the software) in the other. This delay, when taken in conjunction with the controlled variable function, describes a set of relations that differ only with respect to time. Any one of these relationships is valid. In other words, given a delay of 0.2 seconds and the relation:

$$\text{DiffPress} = \frac{\text{FuelLevel} - \mathit{Offset}}{\mathit{Scale}} \times 255$$

really states that the differential pressure device behaves as follows:

$$\text{There exists } \delta \leq 0.2 \text{ such that } \text{DiffPress}(t + \delta) = \frac{\text{FuelLevel}(t) - \mathit{Offset}}{\mathit{Scale}} \times 255$$

This style helps readers focus on the intended behavior by not increasing the complexity of the "idealized" version of the relation. However, it still manages to capture a real-world constraint.

3.9 SPECIFYING ACCURACY CONSTRAINTS

Accuracy constraints describe the way variables behave relative to the inherent or allowed deviation from the ideal. As with timing, the engineer describes an ideal behavior, then provides additional information that precisely captures the permissible deviances from this ideal. For example, the differential pressure unit of the FLMS has the following (ideal) relationship to the fuel level:

$$\text{DiffPress} = \frac{\text{FuelLevel} - \mathit{Offset}}{\mathit{Scale}} \times 255$$

This relationship can vary by $\pm (k_1 + k_2)$, where k_1 accounts for the imprecision of the device's ability to gauge fuel level in terms of pressure (approximately 2 percent), and k_2 takes delays into consideration. This sum is the accuracy constraint that the engineer expresses. He may therefore expect his hardware to always measure the fuel level to within this deviation. Thus:

$$\left| \text{DiffPress} - \frac{\text{FuelLevel} - \mathit{Offset}}{\mathit{Scale}} \times 255 \right| \leq k_1 + k_2$$

3.10 DETERMINING COMPLETENESS AND CONSISTENCY

The Consortium selected models in CoRE, as well as the possible representations, in part because they are amenable to consistency and completeness checking. Because of the formal approach used, these checks have several desirable properties. They do not require a detailed knowledge of the underlying formalisms or detailed knowledge of the domain. They can be performed systematically and yield an unambiguous resolution of consistency or completeness. They have or can be extended to a formal basis that is potentially automatable. Finally, they can be applied independently and concurrently. This facilitates a requirements-writing process that catches problems early.

This section presents some of the consistency and completeness checks that engineers can apply when using CoRE. There is no attempt to enumerate all of them. Instead, what is presented gives a feel for the different kinds of checks that are possible and for how a check might be (manually) applied to verify some aspect of a specification.

The examples are occasionally presented in the representation used for the FLMS specification. However, the consistency and completeness criteria are independent of the representation. They derive from properties of the underlying data models and would apply regardless of the representation used. First stated are the criteria in terms of the underlying models. Then how the criteria are applied using a specific representation is shown.

3.10.1 Data Model Completeness Checks

Since the data model reflects the set of parts that constitute a complete specification, many of the completeness checks require simply checking for the presence of certain types of data. The following completeness criteria illustrate this:

- All attributes and data items must have a definition including type.
- All monitored and controlled variables must have a physical interpretation.
- All environmental variables and data items must participate in at least one relationship.

The appropriate technique for checking these criteria depends on the representation of the data. In the case of the FLMS example, the engineer can do it by visual inspection of the graphic or textual representations. For instance, the alarm attribute is defined as:

Alarm	:ENUMERATED: Alarm = sound if the audible alarm is sounding. Alarm = silent if the audible alarm is silent.

The attribute would be incomplete if it lacked either the definition of its type (":ENUMERATED:"), possible values, or the physical interpretation.

The completeness of parts of a specification that an engineer can determine through inspection (or by graph-searching strategies if automation is available) include:

- Every attribute must be associated with an object.
- Each term, event, or mode used in one object specification must be defined locally or on the interface of another object.

These and many more checks are possible for each aspect of the model. For instance, the engineer can easily represent the input and output data items in template form and check them visually or automatically. It is easy to see such checks are feasible, although doing it on a large information model would be tedious. Fortunately, most of these kinds of checks are automatable. This is true for most of the completeness and consistency checks listed.

3.10.2 Data Model Consistency Checks

Consistency checks of the data model help ensure that the information in the various models is not self-contradictory. The engineer performs these kinds of checks by comparing the components of the model. For example:

- Every attribute or data item is defined exactly once.
- There is exactly one function defining the value of each controlled variable or output data item.

3.10.3 Behavioral Model Completeness Checks

The behavior model is complete if the engineer specifies the system's effect on controlled variables for all possible values of the monitored variables. The constructs in the data model that contain this

information are mathematical relations. Whatever the representation, he must determine the completeness by checking that each relation covers all values in its range and accounts for all values in its domain (i.e., if it never assumes a possible value in the domain, the specification should say so).

Completeness checks in the behavior model can fall into two classes: those that can be accomplished by inspection, and those that require analysis of the specifications semantics. The engineer former can check by examining the specification to make sure all necessary information has been included. The presentation of the information in the FLMS specification has been chosen to make such inspections easy. The following are examples of such checks:

- Each mode class must have an initial mode, and every mode must be reachable. In the pictorial representation of the FLMS InOperation mode class, the initial mode is indicated by an arrow without a source. Reachability can be determined by examination of the transition diagram. See Figure 7.

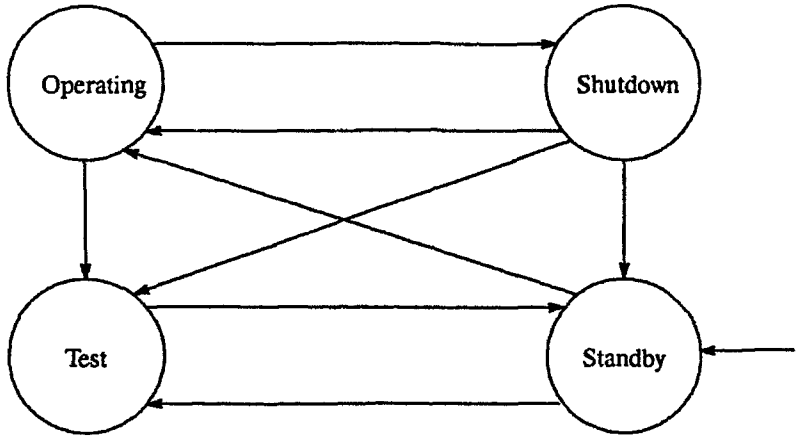

Figure 7. Fuel-Level Monitoring System: InOperation Modes

- The same event cannot cause a transition from one mode to two or more others.
- The event that causes each transition to occur must be specified.
- Each controlled variable function must specify the value for every possible mode of any relevant mode class and must specify the cases corresponding to each possible controlled variable value (even if the case is "never").

There are also analysis-based checks that require examination of the entire model. For example:

- Each monitored variable must appear in at least one relation or mode transition.
- Each monitored (controlled) variable must appear in at least one relation in IN (OUT).
- Each input (output) data item must appear in at least one relation in IN (OUT).

Other checks relating to parts of the specification given in term of state machines or functions are more localized. Consider the relations that define controlled variable behavior, specifically those specified as piece wise-continuous functions partitioned on conditions and modes. The conditions are

defined in terms of monitored variables. The set of conditions and modes partition the functions domain; hence, they must cover all possible states of the monitored variables. It follows that a function is incomplete if all its conditions together do not cover the entire domain. Similarly, if a function is partitioned based on modes of some mode class, it is complete only if the partition covers all modes in the class. Consider the LevelDisplay function, which has the following definition:

$$\text{LevelDisplay} = \begin{cases} \text{Round(FuelLevel, 1)} & \text{if in Operating, Shutdown or Standby mode} \\ 0.0 & \text{if in Test mode and } 0 \le \text{Time} - t_0 < 4 \\ \lfloor(\text{Time} - t_0\rfloor \times 11.1 & \text{if in Test mode and } 4 \le \text{Time} - t_0 < 14 \\ 0.0 & \text{if in Test mode and } 14 \le \text{Time} - t_0 \end{cases}$$

This function is partitioned based on a combination of modes and conditions. By examining the information in the right column, it can be determined that the partition based on modes is complete since all four modes in the class are included. The function is further partitioned in Test mode based on a set of conditions that divide the function over all possible values of the monitored variable. It is therefore complete with respect to its domain.

Note that the function has actually been over-specified, since it is known by the mode transitions that the condition $\text{Time} - t_0 > 14$ seconds is impossible. The condition is included so that the function's completeness can be verified.

3.10.4 Behavioral Model Consistency Checks

The aim of consistency checks is to find contradictions, so by their nature they are not template-based. Instead, they require some analysis of certain portions of the requirements. The formal structure of the data model in CoRE constrains the range of places where a given inconsistency can occur. That is, checking the consistency of a given component generally means searching a fixed set of places for specific conditions rather than searching the entire requirements specification.

The essence of consistency checking is to determine if any relation contains a contradiction. As with the completeness checks in the externally visible behavior model, the types of analyses range from simple checks for the presence of information to verification of certain mathematical properties. The following are examples of such checks:

- The value assigned to a controlled variable must be consistent with the variable's data type. For instance, the function that controls the Watchdog timer in the FLMS correctly assigns it a value drawn from the domain TIME. This is verified by determining the function that assigns a value to the variable and by ensuring that all pieces of the function result in a value of the correct type.

- The delays associated with the hardware must not exceed those specified as required behavior. More precisely, suppose a function F monitors *m* and controls *c*. Let *i* be the input data item associated with *m* (i.e., there exists a mapping in IN whose domain is *m* and whose range is *i*), and *o* be the output data item associated with *c*. If F has an associated delay *d* specified, then the sum of the delays in the related IN and OUT mappings cannot exceed *d* (if it does, the hardware is too slow to meet the system requirements). This is verified by determining all delays associated with functions in REQ and then, for all such delays, locating the relevant data items and checking the sum of their delays. For instance, in the FLMS the delay for

HighAlarm is ShutdownLockTime/2 - 1ms; the differential pressure unit has a delay of 0.2 seconds, and the alarm a delay of DisplayDelay. Therefore, this criterion is satisfied when the relation ShutdownLockTime/2 - 0.001 < *DisplayDelay* + 0.2 holds.

- The domain of the NAT relation must be a subset of the domain of the REQ relation (otherwise there are naturally occurring states that a system cannot handle). This is in some sense the inverse of the completeness check requiring that a function's domain be complete with respect to the variables it monitors. Functions written this way, guarantee consistency since the domains of variables being monitored are a superset of the domain of NAT. This consistency property is therefore checked by making sure each function completely covers its domain and that all monitored variables and mode classes are used in some function.

- In a relation represented using conditions, exactly one of the conditions must be true for any possible set of values of the conditions (i.e., the external state). In other words, each function's value must be unambiguously determinable. Consider the LevelDisplay function above, the value of which in Test mode is represented by the following condition table:

Condition	**LevelDisplay =**
$0 \leq \text{Time} - t_0 < 4$	0.0
$4 \leq \text{Time} - t_0 < 14$	$\lfloor \textit{TestTime} - 4 \rfloor \times 11.1$
$\text{Time} - t_0 \geq 14$	0.0

By inspection, it can be seen that none of the conditions in the left column overlap, which satisfies the consistency property desired.

- The mode transitions must be deterministic so that each event can be associated with only one transition from a given state. Thus, given an event, the transition to occur, if any, is uniquely defined. This can be verified by identifying all modes with more than one outgoing transition. For each such mode, the conditions with the associated transitions must be disjoint. In Figure 7, for example, there are two transitions emanating from the Operating mode. The events triggering them are:

  @F(*LowFuelLimit* < FuelLevel < *HighFuelLimit*)
  WHEN [SlfTst ≠ pressed]

  @T(DURATION(SlfTst = pressed) ≥ 0.5s)

Since the first event can only occur when the self-test button is not pressed, and the second when it is, these clearly do not represent the same event.

This page intentionally left blank.

4. METHODS OF REPRESENTATION

The method requirements state that the basic method should not assume any particular documentation standard or format. Nonetheless, the Consortium's work must address some issues of representation to satisfy the complete set of method requirements. For instance, the Consortium must show that the method can be applied using common, existing notations. It must show that ways exist of representing and presenting software requirements produced by the method exist that are understandable to a variety of audiences. Finally, it must produce realistic examples that it can demonstrate satisfies these method requirements and the guiding principles, so it must choose particular organizations and notations. This section, discusses issues of representation, describes the formats and notations used in the examples, and discusses possible alternatives. Subsequent work will provide explicit guidance for using CoRE with a variety of common notations and mapping the work products to company or other standards (e.g., DOD-STD-2167A).

For any requirements specification, there will be several audiences with distinct needs (e.g., customers, systems engineers, software engineers, testers, and project managers). The Consortium's assessment of the member company problems leads to the conclusion that there is not one representation of the requirements that will adequately serve the needs of every one of the audiences. Some readers want an overview or introduction to the software. Others want precise answers to specific questions that they have about what the software must do or how quickly it must do it. Representations of requirements information must serve not only to answer questions about the system being specified, but must also serve the creation and recording of the information and the verification of its consistency and completeness.

The backgrounds of those in the audience vary widely. Some have been trained and are experienced in computer science and related disciplines. Some are engineers trained and experienced in traditional engineering disciplines. Some have been trained and are experienced in areas other than computer science and engineering.

In addition to variety in the users of a requirements specification, there is a wide variety in the standards to which organizations using a Consortium-produced requirements method must adhere. Different member companies follow different standards for software development processes, methods, notations, and document organizations. Government customers impose many of the standards. Member companies or organizations within member companies impose others.

The Consortium uses a two-part approach to address this variety in the audience of the requirements specification and in the standards to which it must adhere. First, it defines a requirements data model that defines the underlying content and organization of requirements information (Section 2.2). Second, it shows how different views or representations of information in the model can serve the needs of different audiences. This section illustrates some techniques for representation requirements that satisfy the method requirements (e.g., serve different types of users). Future work will provide guidance in mapping the model and methods of representation to specific standards.

4.1 REQUIREMENTS INFORMATION TO PRESENT

This section describes different collections of types of requirements information that are useful to present to requirements specification users. Each of these collections may be thought of as a view of the requirements. These views are discussed in the following order:

- The environment of the system
- The hardware/software interface
- The behavior of the system
- States of the system and state transitions
- Timing and accuracy constraints

A description of the environment of a system can serve as an introduction to or an overview of the system. Such a description should include characterizations of relevant classes of objects in the environment and important relationships among the classes and objects. This information is recorded by the following entities in the requirements data model:

- Environment variables
- Objects, attributes, and relationships in the information view
- The NAT relation

Users of the requirements specification must be able to distinguish between behavior that the software is responsible for accomplishing and behavior that is the responsibility of the hardware. Those who are writing the software and those who are designing the hardware need to know what facilities the hardware is providing and how to use them. This information is provided by the hardware/software interface which is recorded by the following information:

- Input and output data items
- The IN and OUT relations

Those who are designing and writing the software, those who are testing it, and many of those who use it need to know exactly what the software must do. Some users and others need an informal indication of what the software must do. All these needs can be served by a description of the behavior of the system, which is provided by the following information:

- Environment variables
- The REQ relation

The behavior of a system often depends on the system's history, i.e., what has happened to it in the past. Modes (classes of system states) characterize the history of the system. Those who need to know exactly what the system must do often need to know what modes the system can be in and what can cause it to change modes. This view of requirements information is provided by:

- System modes
- Mode transitions

Designers, writers, and testers of the software need to know how often certain actions must be performed, how long the system may take to respond to certain events, and how accurate its responses must be. This view is provided by:

- Timing constraints
- Accuracy constraints

4.2 PRESENTING REQUIREMENTS INFORMATION

The previous section discussed the information that needs to be presented to users of the requirements. This section discusses and illustrates with examples different ways in which the information can be presented. The examples are intended to be neither definitive nor exhaustive. For all the example presentations there are alternative notations, some of which may be preferred in certain situations.

4.2.1 Presenting the Environment of the System

There are a number of ways in which users of the requirements can be presented with a representation of the environment of the system. The user can be presented with an information view of the environment, that is a presentation of the static structure of the environment. Figure 8 uses an entity relationship attribute (ERA) diagram to present the information view. An ERA diagram represents the entities (or classes of objects) in the environment and the relationships among and attributes of the entities. In the figure, rectangles represent entities. The name of the entity is the name in bold in the rectangle. The other names in the rectangle represent the attributes of the entity. Diamonds connected by line segments to entities represent relationships among the connected entities. The diamond contains the name of the relationship. The number annotating the relationships indicate the maximum and minimum number of instances of each entity that may participate in the relationship. In the example, for each entity participating in one of the relationships, there needs to be exactly one instance of the entity. The diagram is drawn so that a sentence can be constructed from the names of a relationship and the entities to which it relates by reading either top-down or left-to-right. For example, operator operates FLMS and FLMS monitors Fuel in Tank.

Figure 9 illustrates another information view. The diagram presents an example of the domain-independent subtype relation IS A. Both Host and Potential Threat are subtypes of Aircraft. Note the annotation 0:N on the tracks relation. It indicates that Host may track zero or more Potential Threats.

By itself, the information view can provide an informal understanding of the system. The attribute definitions can help provide a more precise understanding. Table 2 illustrates several.

Table 2. Example Fuel Level Monitoring System Attribute Definitions

Attribute	Definition
PumpSwitch	:ENUMERATED: If PumpSwitch = closed then the contacts for switches S1 and S2 are closed. If PumpSwitch = open then the contacts for switches S1 and S2 are open.
WDTimer	:TIME: Time, in seconds, until the Watchdog system assumes that the FLMS has failed.
FuelLevel	:LENGTH range 0.0 .. 30.0: Level of fuel in the tank, in centimeters (cm), along the vertical axis on the left side of the tank, 5 centimeters from the front edge. The level is measured with respect to the scale.

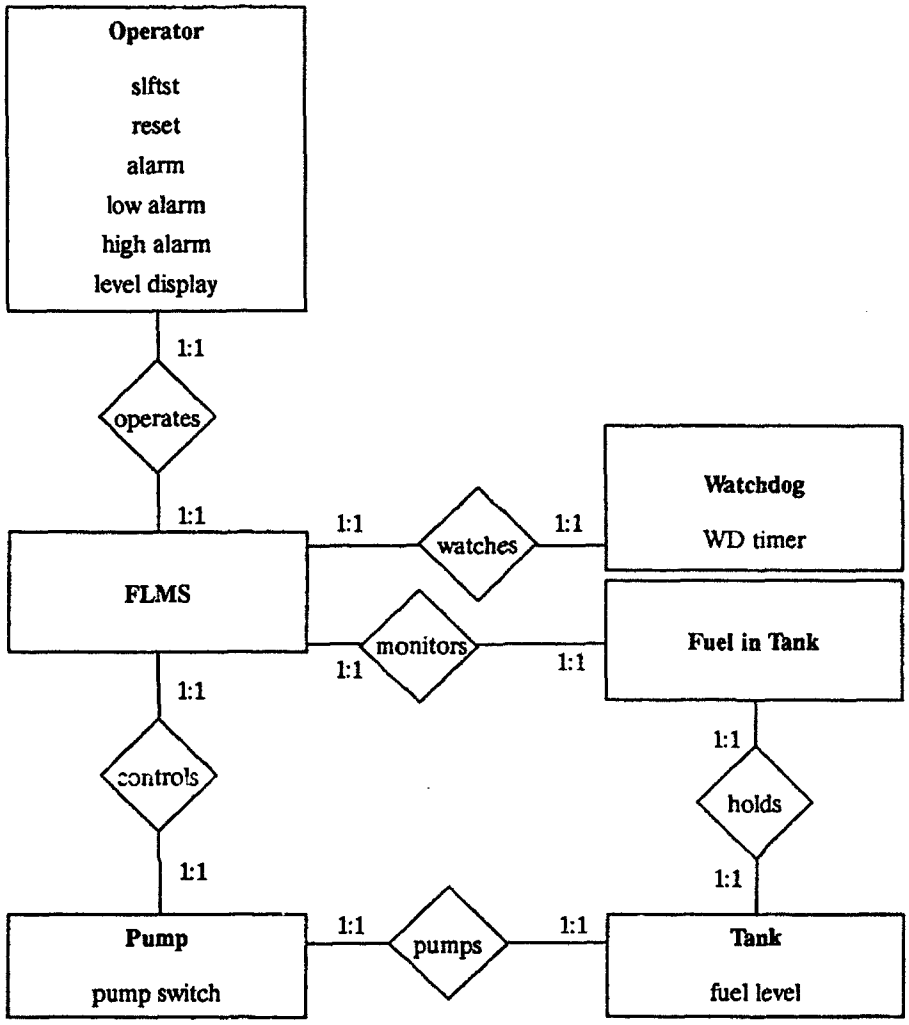

Figure 8. Fuel-Level Monitoring System: Information View

A structured analysis context data flow diagram[3] provides another useful presentation of the environment of the system (see Figure 10). The information that the diagram presents includes the relation between environment (monitored and controlled) variables and the entity with which each is associated. In the example, the circle at the center represents the system (FLMS). The rectangles represent entities in the environment. A labeled arrow from an external entity to the system represents the monitored variable with that name. A labeled arrow from the system to an external entity represents a controlled variable with that name. The dashed arrow to Watchdog indicates that the value of the controlled variable WDTimer is not continuously available.

As with the information view, the context diagram by itself can provide some informal understanding of the system. A more precise understanding of the system can be obtained by referring to definitions of the environment variables (see Tables 3 and 4). These definitions can also serve as presentations of the environment of the system. They represent variables in the environment whose values the system must set (the controlled variables) and variables in the environment whose values the system must monitor (the monitored variables) to determine the desired values of the controlled variables.

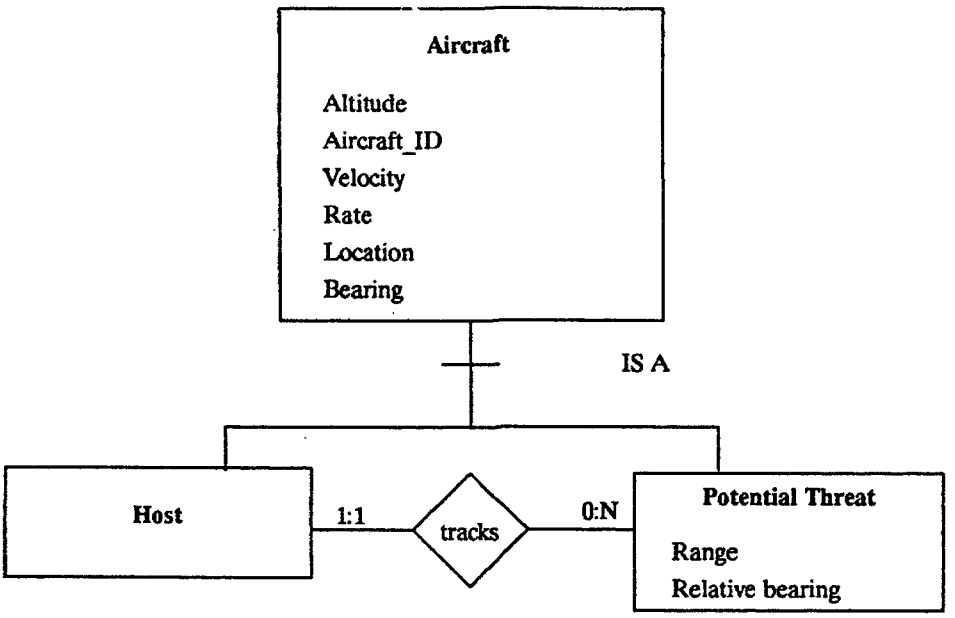

Figure 9. Aircraft Collision Warning Monitor: Information View

Table 3. Definitions of Monitored Variables

Monitored Variable	Type	Definition
Reset	enumerated	Reset = pressed iff the push button labeled RESET is pressed. Reset = released, otherwise.
FuelLevel	length	Range 0.0 .. 30.0. Level of fuel in the tank, in centimeters (cm), along the vertical axis on the left side of the tank, 5 centimeters from the front edge. The level is measured with respect to the scale.

Table 4. Definitions of Controlled Variables

Controlled Variable	Type	Definition
Alarm	enumerated	Alarm = sound iff the audible alarm is sounding. Alarm = silent iff the audible alarm is silent.
PumpSwitch	enumerated	If PumpSwitch = closed then the contacts for switches S1 and S2 are closed. If PumpSwitch = open then the contacts for switches S1 and S2 are open.

3. Structured analysis notation is used in several of the presentations. The interpretation of the meaning of the notation (which is explained in the text) varies from the traditional structured analysis interpretation.

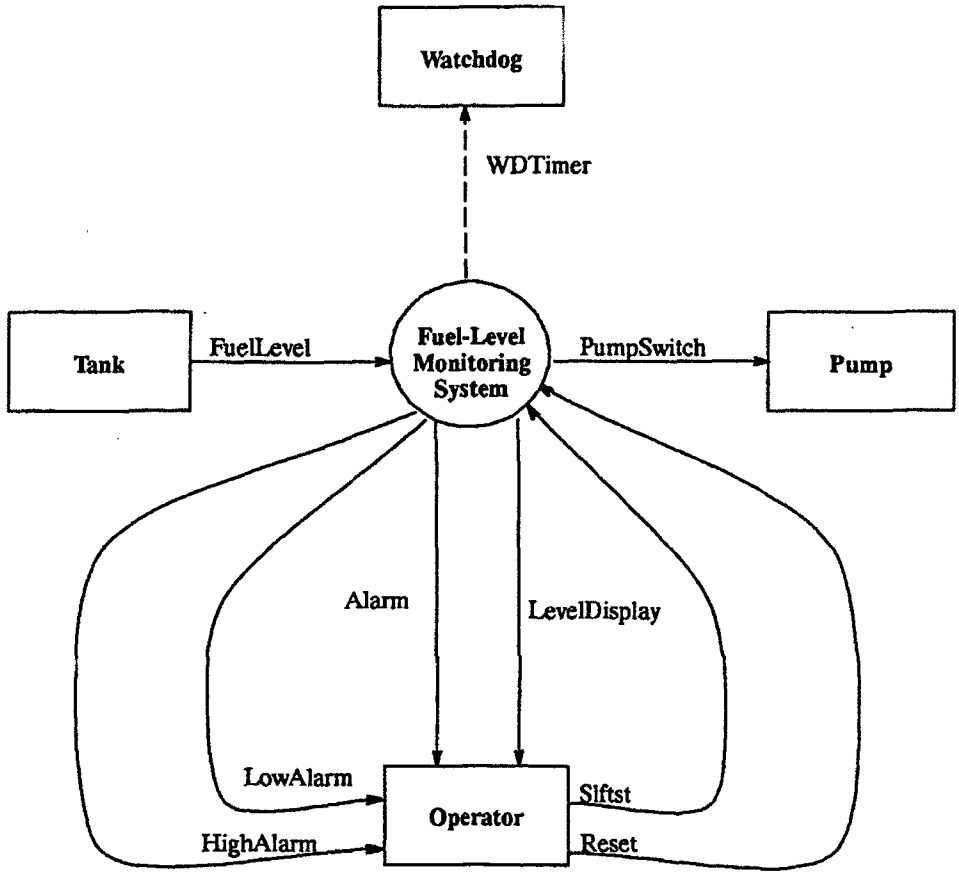

Figure 10. Fuel-Level Monitoring System: Context Diagram

4.2.2 Presenting the Hardware/software Interface

One way of presenting the hardware/software of the system is to use the structured analysis notation that was used to present the environment of the system augmented with a notation to represent devices (see Figure 11). In this case, the circle in the middle of the diagram represents the FLMS software (when representing the environment of the system, the circle represented the entire FLMS system, hardware and software), the rectangles represent entities in the environment of the system (as before), and the parallelograms represent hardware devices that are part of the FLMS. A labeled arrow from a device to the software represents an input data item on the interface between that device and the software. The software can read the input data item and the device can control it. A labeled arrow from the software to a device represents an output data item on the interface between that device and the software. The software can set the output data item and the hardware can read it. Labeled arrows between the external entities and the devices represent environment variables. An arrow from an external entity to a device represents a monitored variable whose value the device can sense. An arrow from a device to an external entity represents a controlled variable whose value the device can control.

As is often true of graphical presentations, this view of the hardware/software interface provides a useful overview, but it does not constitute a complete specification. The information needed to make a complete specification of the hardware/software interface can be provided in the form of tables and templates. These tables and templates represent another view of the interface.

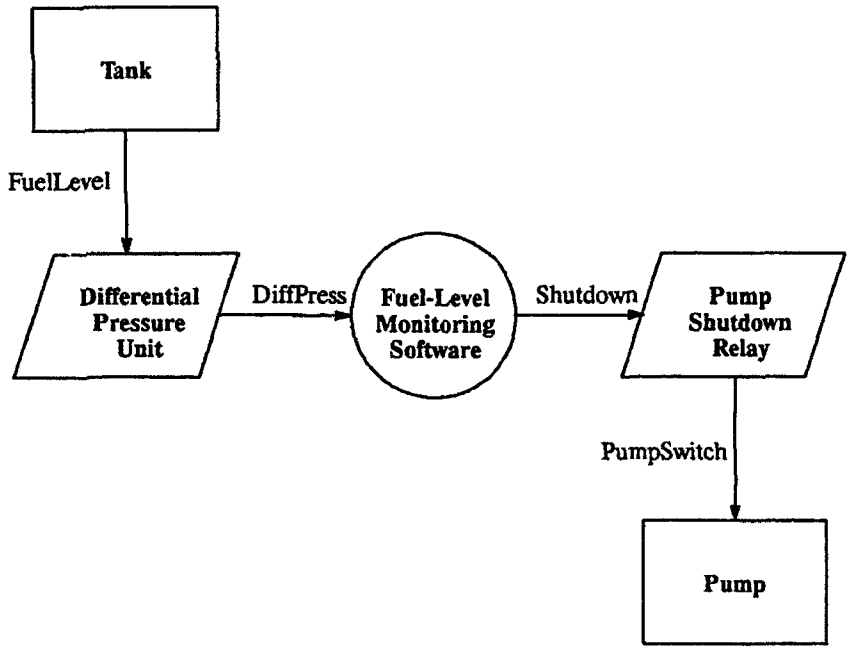

Figure 11. Fuel-Level Monitoring System: Software Context Diagram (Partial)

An example representation of the information that defines an input data item is as follows:

Acronym: DiffPress

Hardware: Differential Pressure Unit

Characteristics of Values:

Values: DiffPress $\in$ [0, 255]

Data Transfer: ADC(0)

Data Representation: 8-bit unsigned integer

The form of the representation is a template.

An example representation for the output data item Shutdown is as follows:

Acronym: Shutdown

Hardware: Pump Shutdown Relay

Characteristics of Values:

Value Encodings:	operate	(1b)
	shutdown	(0b)

Data Transfer: PortC

Data Representation: Bit 1 of byte

The set of similar definitions for all the input and output data items represents a detailed view of the hardware/software interface. This view can be made complete by representing the IN and OUT relations. The part of the IN relation that specifies the value of DiffPress that the hardware is responsible for maintaining is represented as the decision table illustrated in Table 5. A decision table is divided into two parts, the condition or top part, and the decision or bottom part. A condition or decision that holds is indicated by marking it with an X. The column in which a mark appears is significant. Marking a decision means that the decision holds while all of the conditions that have marks in the same column as the decision's mark are true. If two or more conditions are marked in the same column, then the decision that is marked for the column holds while the conjunction (and) of the marked conditions holds. It can be seen from the table that if the fuel is within its calibration bounds, then the value of the input data item DiffPress is a function of FuelLevel and the constants Offset and Scale (whose definitions have been omitted from this example).

Table 5. The IN Relation for DiffPress

Condition			
LowerCalibrationBound $\leq$ FuelLevel $\leq$ *UpperCalibrationBound*	X		
FuelLevel < *LowerCalibrationBound*		X	
FuelLevel > *UpperCalibrationBound*			X
Decision			
DiffPress = ((FuelLevel – *Offset*) / *Scale*) x 255	X		
DiffPress = 0		X	
DiffPress = 255			X

4.2.3 System Behavior

4.2.3.1 Presenting an Overview

There are a number of ways in which an overview of the system's behavior can be presented. Such overviews do not serve as specifications of the behavior. Rather, they are intended to serve as introductions, reminders, and pointers. Several alternative presentations are:

- Real-Time Structured Analysis (RTSA) context and level 1 diagrams.
- Controlled variables as text or a table.
- Text or table abstraction of controlled variable functions.

RTSA data flow diagram notation can present an overview of the system's behavior. A high-level view is provided by the context diagram that was discussed earlier (see Figure 10). Figure 12 shows a more detailed view. The circles represent objects that encapsulate parts of the requirements. An object that encapsulates a mode class is distinguished by its dashed outline.

The interpretation of an arrow depends on whether the arrow connects one object to another or connects an object to an entity in the environment (such entities are represented in the context diagram only). Arrows between objects represent information about monitored variables. The information is

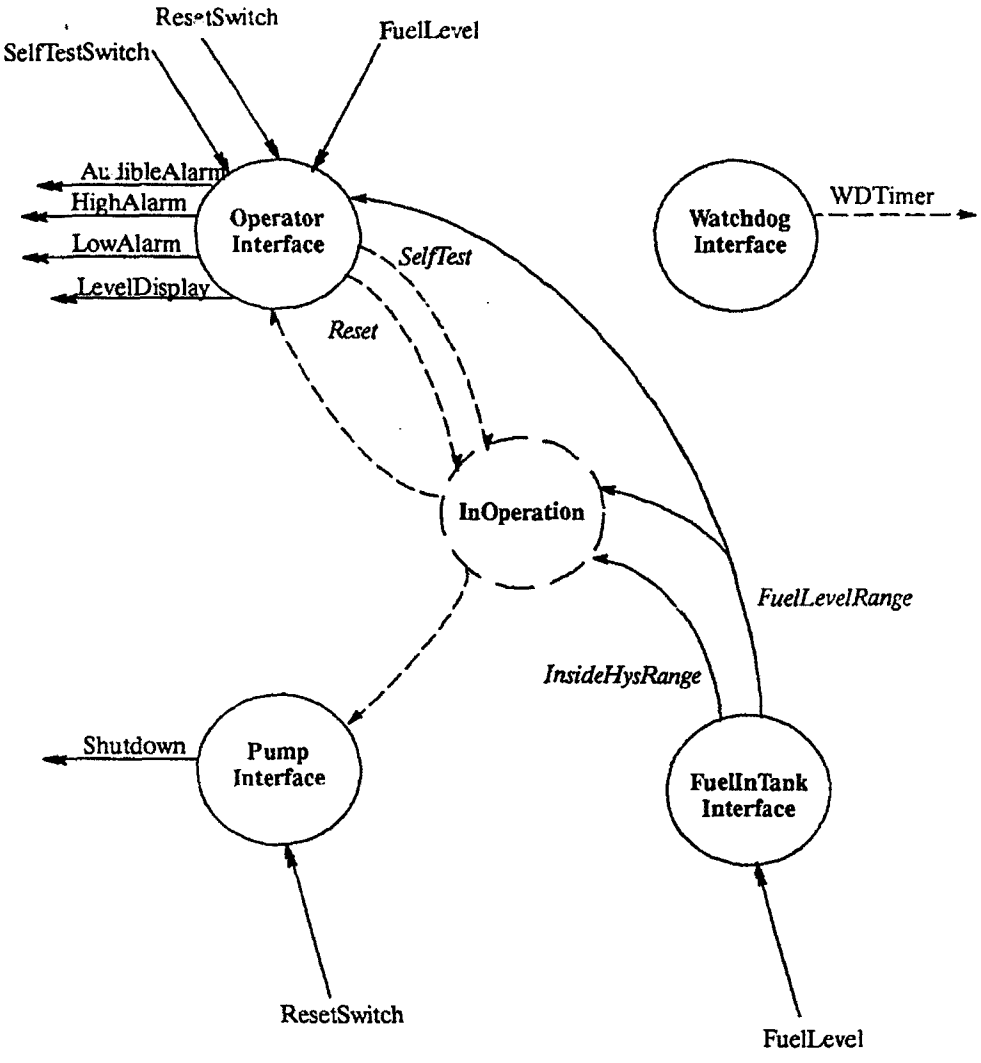

Figure 12. Fuel-Level Monitoring System: Transformation Diagram

on the interface of the object at the tail of the arrow. The object on the head of the arrow uses the information. Solid arrows between objects represent conditions, and dashed arrows between objects represent events.

An arrow from an entity in the environment to an object represents a monitored variable. An arrow from an object to an entity in the environment represents a controlled variable. A dashed arrow represents a variable whose value is not continuously available. A solid arrow with a double arrowhead represents a variable whose value is continuously available.

The table of controlled variables in Table 4 represents another overview of the system's behavior.

4.2.3.2 Detailed Specification of the System's Behavior

The REQ relation specifies the system's behavior. Typically, a set of functions define REQ, each of which specifies the behavior of one or more controlled variable. There are a number of ways in which

we can specify these functions. We can use conventional mathematical notation. (See, for example, the specification of LevelDisplay in Section 3.5.1.)

A decision table can represent a controlled variable function. Such a representation is useful if the value of the function depends upon a number of conditions, as is the case for LevelDisplay in Section 3.5.1. A decision table representation of the LevelDisplay function is illustrated in Table 6 (Section 4.2.2 describes how to read a decision table). It can be seen from the second column of the table (the first column of X's) that if the system is in one of the modes Operating, Shutdown, or Standby that:

$$\text{LevelDisplay} = \text{Round}(\text{FuelLevel}, 1)$$

Table 6. Decision Table Representation of the Behavior of LevelDisplay

Condition				
in mode Operating, Shutdown, or Standby	X			
in mode Test		X	X	X
$0 \leq \text{Time} - t_0 < 4$		X		
$4 \leq \text{Time} - t_0 < 14$			X	
$14 \leq \text{Time} - t_0$				X
Decision				
LevelDisplay = Round(WaterLevel, 1)	X			
LevelDisplay = $\lfloor \text{Time} - t_0 \rfloor \times 11.1$			X	
LevelDisplay = 0.0		X		X

A tabular alternative to the decision table is the condition table (see Table 7). The condition table can be thought of as a decision table that has been modified to better support consistency and completeness checking (see Section 3.10) and to be more concise. The first column of the table (labeled Mode) defines the modes that apply to each of the rows. The other columns correspond to distinct expressions that specify values for the controlled variable. These expressions appear in the bottom row of each column. The entry in the bottom row of the first column (LevelDisplay =) serves as a reminder of the meaning of the expressions in the bottom row. An entry in one of the columns labeled Condition defines the condition in which, when the system is in the mode identified in the first column of the row, the expression at the bottom of the column specifies the required value of the controlled variable.

Table 7. Condition Table Representation of the Behavior of LevelDisplay

Mode	**Condition**			
Operating OR Shutdown OR Standby	always			
Test		$0 \leq \text{Time} - t_0 < 4$	$4 \leq \text{Time} - t_0 < 14$	$14 \leq \text{Time} - t_0$
LevelDisplay =	Round(WaterLevel, 1)	0.0	$\lfloor \text{Time} - t_0 \rfloor \times 11.1$	0.0

For example, "always" in the first condition column indicates that whenever the system is in one of the modes Operating, Shutdown, or Standby:

$$LevelDisplay = Round(FuelLevel, 1)$$

The third condition column indicates that if the system is in Test mode and $4 \leq \text{Time} - t_0 < 14$, then:

$$LevelDisplay = \lfloor Time - t_0 \rfloor \times 11.1$$

4.2.3.3 Presenting States of the System and State Transitions

Where required behavior depends on the order of events in time, mode classes and modes are used to capture the ordering. A mode class is a finite state machine defined on the states of the system's monitored variables. Each state of the machine, called a mode, corresponds to a set of system states. Since events describe the state transitions, changes in the system state, the machine "inputs" are events.

Modes and mode transitions can be presented using any of the techniques used to present finite-state machines. Some commonly used techniques are state transition diagrams, state transition tables, and state charts. Figure 13 illustrates a state transition diagram representation of the InOperation mode class from the FLMS. A circle labeled with the name of the mode represents each mode in the mode class. The unterminated arrow to the Standby mode indicates that Standby is the initial mode. Arrows labeled with the event that causes a transition indicate transitions between modes. In the example, the expressions defining the events are too complicated to show in the diagram. Technical terms are used instead. Table 8 shows the event that each term represents.

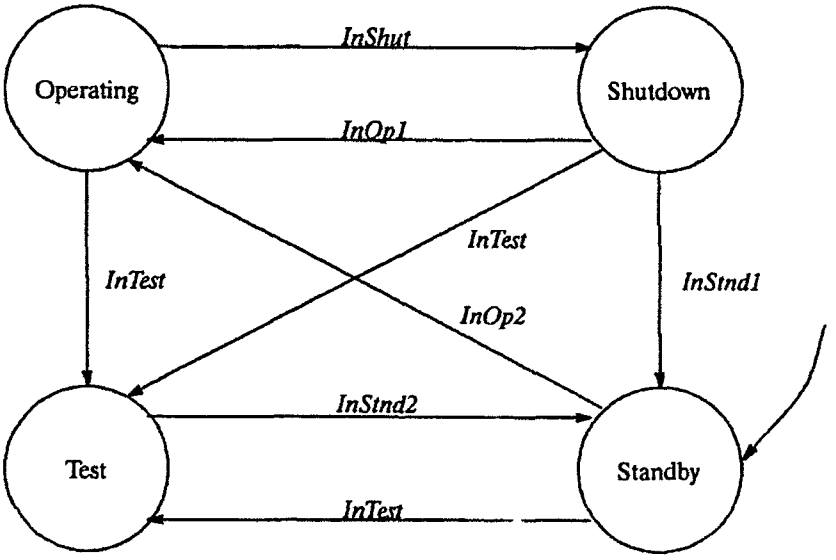

Figure 13. Fuel-Level Monitoring System: Operating Modes

Table 8. Definitions of InOperation Technical Terms

Term	Definition
InOp1 =	@T(*InsideHysRange*) WHEN (DURATION (INMODE (Shutdown)) < *ShutdownLockTime*)
InOp2 =	@T(*Reset*) WHEN (*InsideHysRange*)
InShut =	@F(*FuelLevelRange* = *WithinLimits*)
InStnd1 =	@T(DURATION (INMODE (Shutdown)) > = *ShutdownLockTime*)
InStnd2 =	@T(DURATION (INMODE (Test)) = 14s)
InTest =	@T(*SelfTest*)

The same mode class is represented as the state transition table in Table 9. The indication that Standby is the initial state would be provided by some means other than the table. The first column of the table lists the "current" mode of the mode class. The row at the bottom of the table lists the "new" mode. An entry in the table is the event that causes a transition from the mode in the first column of the row to the mode in the last row of the column. The same technical terms that were used in Figure 13 are used in the table. An X indicates that the mode class cannot transition from the current mode to the new mode.

Table 9. State Transition Table Representation of the InOperation Mode Class

Mode	Triggering Event			
Operating	X	*InShut*	X	*InTest*
Shutdown	*InOp1*	X	*InStnd1*	*InTest*
Standby	*InOp2*	X	X	*InTest*
Test	X	X	*InStnd2*	X
InOperation =	Operating	Shutdown	Standby	Test

For example, if the system is in Standby and the Reset event occurs (i.e., the operator presses the reset button for three seconds) while the fuel level is within range, then the system transitions to Operating. If the SelfTest event then occurs (i.e., the operator then presses the self-test button for at least 500 milliseconds) the system will transition to Test mode. After 14 seconds in Test (the definition of InStnd2) the system transitions to Standby. This example can also be used to trace through the state transition diagram in Figure 13.

5. TECHNICAL RATIONALE AND PROGRESS

This section discusses the reasons behind the technical approach by relating the project's technical direction to the method requirements (discussed in Section 1). It discusses the basic technical approach and how that approach relates to particular method requirements. It also discusses the state of the current work in two parts: first, the state of CoRE itself relative to a complete, practicable method (e.g., how complete the method is); second, how CoRE products satisfy the method requirements (e.g., completeness of a specification produced by CoRE).

5.1 THE BASIC TECHNICAL APPROACH

CoRE represents an amalgamation of features, each of which has proven value by virtue of actual, successful use in systems development. The Consortium made a conscious attempt to synthesize the best features of existing approaches rather than to invent anything new. This section describes the prominent features of CoRE and discusses how these features help answer the method requirements. The discussion is also intended to help the reader compare the properties and goals of CoRE to those of existing methods.

5.1.1 Reactive System Orientation

A nonreactive system performs terminating computations, with all the inputs present as the system begins its computation. A reactive system, on the other hand, maintains an ongoing interaction with its environment, during which it receives inputs and produces corresponding outputs on an ongoing, sometimes unpredictable basis. Such systems have very different properties from nonreactive systems, and the requirements definitions must reflect these properties. For example, finite state machines often express the properties of reactive systems well, specifying the effects of inputs in terms of the system's current mode of operation. Since most real-time and embedded systems developed by member companies fit the reactive system pattern, the Consortium took this pattern as a basis for the requirements method.

The reactive system orientation addresses the requirements both directly and indirectly. Directly, the reactive orientation addresses the requirement that CoRE support real-time embedded systems. Indirectly, the reactive orientation is exploited by developing a standard model for expressing system requirements based on characteristics common to reactive systems. As discussed in Section 3, the system is modeled as a state machine interacting with the environment. Requirements are expressed in terms of the ongoing, real-time relations the system must maintain between the environmental variables monitored by the system and those it controls. Using such a standard, formal model allows much of the requirements process to be standardized as well and allows increased rigor in the specification. For instance, it is clear when requirements are incomplete, and there are analytical tests for consistency and completeness.

5.1.2 Front Loading

The requirements model is a detailed description of system structure and behavior which is created at a very early stage of the development life cycle. This implies that significant time will be devoted

to analyzing the model to find errors and inconsistencies in requirements before the code is written. In traditional systems development approaches, many such problems are not found until after the code is written. This front loading of the development process should reduce overall development time, since studies show that an error in the requirements model is easier to fix than the manifestation of the same error appearing in the design and code. However, the implication is that estimates of relative duration of project phases will have to be modified to accommodate the new approach.

5.1.3 Integration of Graphics-Based and Text-Based Methods

Schematic aspects of the requirements model (layout and connection of parts, decomposition of parts into subparts) can be expressed in graphic form with supporting text. The graphic notations have a rigorously defined syntax and semantics, and each graphic element has an alternate textual form. Thus, a model can be expressed totally in textual form, or in a mixture of graphics and text. (The Consortium anticipates that appropriate tools will ultimately be able to capture a model in either form and display or print it in either form as required.) The graphics notations (entity-relationship diagrams, data flow diagrams, state transition diagrams) are outgrowths of Real-Time Structured Analysis and are adapted from the CASE/Real-Time Method (Ward 1989). Parts of the textual notations were adapted from those developed by David Parnas and his colleagues (Henninger et al 1978; van Schouwen 1990).

Integration of graphics-based and text-based methods addresses concerns for improved communication among a variety of audiences. The ability to provide a graphic representation allows readers to quickly grasp essential relationships among system components and gain an overview of its organization. By providing a consistent, formal textual interpretation, the graphics combine smoothly with the detailed specifications best given in text. This allows developers to sketch ideas and do early development in either the visual or textual medium and have the result carry over directly into the formal specification. Finally, the method assumes the use of common data flow notations (e.g., Ward/Mellor), which satisfies the requirement for not inventing unnecessary new notations. The Consortium will write future guidebooks for CoRE to allow the use of a variety of available notations (e.g., Ward/Mellor, Hatley/Pirbhai, CASE Real-Time, Modecharts, or Statecharts).

5.1.4 Integration of the Object-Oriented Paradigm

CoRE expresses requirements in terms of classes of objects in the system's environment. The heuristics for object identification cluster the monitored and controlled variables into groups, all of whose members are likely to be affected by certain variations in requirements. The identification of objects is an extension of the CASE/Real-Time Method and is compatible with the objective of organizing a model to facilitate likely changes as the system evolves. Since objects of each class encapsulate data about themselves, the requirements model obeys the principles of information hiding and data abstraction.

Use of the object-oriented model supports requirements for ease of change and separation of concerns. Objects provide a mechanism for abstracting detail and encapsulating information. As can be done in the software design, objects hide details in the requirements specification that are likely to change. This limits the effects of changes to a small, well-defined part of the requirements in most cases. Most requirements apply to a small number of objects, and many requirements changes apply to a single object. Thus, readers can study, and maintainers can change, different aspects of the requirements independently. The object model also supports the capabilities of dividing the requirements into separate work assignments and the ability to extend the development of selected parts of the system through the implementation to support risk mitigation.

The final version of CoRE will also make use of hierarchical superclass/subclass relations among classes of objects to handle more complex object relationships than are present in the small examples used thus far.

5.1.5 Nonalgorithmic Specification

The Consortium based its approach on specification of functional requirements as relations between environmental variables. By describing only externally visible behavior using relations between monitored and controlled variables, the method permits nonalgorithmic specification of software requirements. This does not mean that all decisions affecting design and implementation are excluded from the requirements definition; rather, the developer is free to choose the level of detail, restricting the specification to decisions about requirements if desired. For instance, the decomposition into objects and the allocation of particular requirements to particular objects represents decisions about the requirements that will constrain subsequent design and implementation decisions. In particular, the decomposition will affect which requirements are most easily changed and, subsequently, which aspects of the design should be easy to change.

The underlying model that CoRE uses defines all the functional behavior in terms of relations among monitored and controlled variables. This allows the analyst to limit the kinds of decisions that he can make to decisions about requirements (as opposed to design or implementation decisions). If more detailed decisions, including specific algorithms, are actually system requirements, he can add these to the specification as a more detailed resolution of the requirements decisions.

5.1.6 A "Machine-Like" Model

Although the requirements model deliberately omits implementation details of the proposed system, it is interpretable as a "virtual machine" for producing inputs from outputs. This positions the model between traditional software designs and declarative, property-oriented formal methods that describe systems in abstract mathematical terms.

Use of the machine model supports the need for a method that is easy to learn, use, and understand. In general, the machine model is simpler than more formal methods and does not require as much training to read, understand, or assess as do some of the more formal approaches. The model also supports the requirement for explicit definition of the environment and control of system boundaries. The model explicitly identifies and defines the required information in the environment by modeling the environmental variables of interest in terms of monitored and controlled variables. The REQ relation clearly identifies the constraints on the environment imposed by the system while the NAT relation explicitly defines the constraints imposed by the system environment. The definitions of the IN and OUT relations capture the interfaces, including all assumptions about hardware devices (to any level of detail).

Finally, the machine model provides a sound basis for automation. The model has a formal basis that can be made machine interpretable. This should provide a basis for developing such capabilities as automated generation of rapid prototypes, semantic analysis of model (e.g., for deadlock), analysis of timing constraints, and simulation.

5.1.7 Existing Tool Support

The graphic notations are compatible with those provided by various CASE products, ensuring that these products can provide a reasonable level of support for building of requirements models in the

absence of a tool optimized for CoRE. For example, the **team***work* products from Cadre Technologies, the Software through Pictures products from Interactive Development Environment, and the Excelerator products of Index Technology Corp. use common notations (from structured analysis) identical or very similar to those used in the examples. Current work is focusing on providing a complete description of applying the method on **team***work*. This section discusses this in more detail later. The Consortium will address additional platforms as demand exists.

This aspect of the approach supports the requirement that the Consortium consider prior member company investment in existing tools.

5.1.8 Document and Work Product Independence

In accordance with the expressed wishes of the member companies, the Consortium developed CoRE to be independent of existing standards for project documentation such as DOD-STD-2167A. Rather than defining the work products in terms of documents or the process in terms of sequences of documentation, CoRE focuses on the information that constitutes a requirements specification. The requirements data model captures the structure of this information. The data model can be thought of as a database schema for requirements. The Consortium developed it to embody only those assumptions about requirements implicit in the method itself (and not assumptions about how that information should be formatted for a given user or customer). The data model provides the common structural basis for different presentations, including DOD-STD-2167A.

5.2 QUALITIES OF CORE

This section discusses the current qualities of CoRE itself as opposed to the process implied by CoRE or its work products; for example, how complete the method description is at this time, how well it does scale-up, and so on. It discusses how far along the work is in satisfying the overall member company needs and identifies ongoing development work.

5.2.1 Completeness of the method

CoRE is complete to the extent that a workable method is available that satisfies the requirements set by the member companies. The project has developed the technical foundation for such a method. The theoretical basis of CoRE is sufficient to encompass the specification of all behavioral requirements for real-time embedded applications. The Consortium has demonstrated the practicality of CoRE for small applications and is currently investigating scale-up. It has prepared an overview of how the requirements definition process will be conducted when using CoRE, but significant process definition work remains.

The following are the areas in which additional development work on the technical approach of the method will be necessary:

- Specification of user interfaces (especially graphical user interfaces) in a manner that potential end users can review.
- Description of display and report formats.
- Specification of accuracy and precision requirements.

The Consortium anticipates refinements to CoRE, and the detailed process for applying it (i.e., a guidebook), during Member Company pilot projects.

CoRE does not require a specific notation, although this report uses one notation consistently. The Consortium is evaluating alternative notations and will present them in subsequent project deliverables. For example, to express information on system states, CoRE may use the statechart graphical formalism as an alternative to the operating modes diagram in the example (Figure 16). Modecharts (Mok 1991; Jahanian, Lee, and Mok 1988) offer additional formal mechanisms for evaluating real-time properties implied by a specification as well as some available tool support.

The requirements for CoRE state that it must also support the description of software requirements for command and control systems, in addition to the real-time embedded system domain. Extensions to the method for the C^2 domain are likely to be required in the underlying system model, reflecting the essential role of a large database and a large amount of structured data, and in the methods used to specify system performance and capacity requirements. The Consortium will address these issues in 1992.

5.2.2 Scalability

The current work only partially addresses scalability, but the basic technology developed to date will scale to software systems of the size being built in the member companies. The most important reason for this confidence in CoRE's scalability is the use of existing technology, on realistic-size applications in developing the technical foundations of CoRE.

To demonstrate the scale-up potential, the Consortium needs a larger example in the embedded system domain. The FLMS example given in this report is useful for illustrating the basic approach but is not sufficiently complex to illustrate the potential for scale-up even where solutions already exist. In particular, the FLMS examples does not illustrate the following scalability issues, even where CoRE provides some capabilities:

- ***Determining the Objects and Their Attributes.*** In the present example the objects and attributes appear "intuitively obvious" and their identification does not adequately demonstrate the criteria used for decomposing a problem into objects.

- ***Monitoring and Controlling Several Object Instances Concurrently.*** CoRE defines the required control in terms of multiple, concurrent finite state machines. Several real-time specification methods including Boeing/Hatley, Ward/Mellor, Shlaer/Mellor, and the CASE/Real-Time Method, have used this approach successfully, so it is clear that the basic approach will scale. However, the FLMS example does not demonstrate this capacity since there is only one common state machine of interest.

- ***Appearance and Disappearance of Objects During System Execution.*** In many systems, the set of object instances is not stable during execution. For instance, a fire control system must track a continuously changing set of targets. The FLMS example does not illustrate this, but CoRE handles it by declaring classes of objects and indexing the instances in the specification. Section 4 discusses this briefly.

- ***Specifying More Complex, State-oriented Behavior.*** Although not illustrated by the current problem, a number of real-time methods have used concurrent state machines to effectively capture state-oriented behavior. Further, it has been shown (Drusinsky and Harel 1984) that a model

based in concurrent state machines is exponentially more powerful than a simple state machine model (alternatively, the specification is exponentially simpler). These suggest that CoRE will scale up. The Consortium is currently investigating more sophisticated graphic techniques such as Harel's statecharts (Harel 1984) for illustrating complex state behavior. It is also investigating methods such as Mok's modecharts (Jahanian, Lee, and Mok 1988) that provide an underlying formal model for simulating or reasoning about absolute timing behavior.

Other issues of scaleability remain to be treated. For instance, it is likely that the Consortium will need to define much more complex relations between the monitored and controlled variable values. Work is also under way in addressing the decomposition of a large system into a set of cooperating subsystems. The method notation and process will support such an approach using multiple models of different domains. In addition, CoRE will need to address scaleability of the graphic techniques used for showing information models and data flows.

5.2.3 Usability on Line Projects

Although CoRE will be understandable and usable by software staff and many systems engineers in the member companies, it has not been used to date outside of the development group. However, CoRE is based on combining techniques that have been successfully applied on real-time, embedded system projects similar to those done in the member companies. Additional method development work needs to be done before member companies can use CoRE to define the requirements for command and control applications. Pilot projects to demonstrate the usability of the method on real-time, embedded system applications will begin before the end of 1991.

5.2.4 Technical transfer

The Consortium has invested a major effort to make CoRE notation more understandable than the notations for other requirements methods based on a mathematically-defined system model (i.e., "formal" methods). CoRE includes diagrams designed to provide system overviews and to facilitate access to reference-oriented material. All graphic symbols have textual equivalents, so the engineer can use the preferred notation. In particular, graphic presentation in the form of an information model or data flow diagram provides a basis for resolving issues with the customer at all stages of development. The common, underlying formal model also links the graphics to the specifications of external behavior. This allows the developer to move between the graphic interpretation, the output functions, and user scenarios, helping to ensure a common understanding between customer and developer.

Documents can derive from the requirements database to meet the needs of different audiences, ranging from customers/end users to software designers. The Consortium is prototyping a hypertext-oriented CASE tool to facilitate development, access, and updating of the requirements database. Pilot projects will provide the definitive assessment of the understandability of the method results and their value as a communication vehicle between the diverse audiences involved in the requirements process.

5.2.5 Document Production Independence

An important project goal is to support the production of documents from the requirements database in a variety of documentation formats. The common underlying data model (based on the object organization) makes this possible. The next example will include the repackaging of the results in a form compatible with DOD-STD-2167A. Future work will attempt to exploit existing document generation

capabilities in the member companies such as tools that generate DOD-STD-2167A documentation from an existing tool database.

5.3 QUALITIES OF THE METHOD WORK PRODUCTS

The Consortium has chosen particular features of CoRE to address particular method requirements such as the ability to analyze specifications for completeness or consistency. This section discusses the rationale for these method design decisions and how these relate to the method requirements.

5.3.1 Completeness of Specifications

CoRE must support analysis of specifications for internal completeness. A specification is internally complete if every part of the specification is fully defined, so that answers to all questions that a reader might have about the system's behavior are available in the specification. In contrast, external completeness refers to whether or not the requirements state all behavior that the customer desires. Thus, analysis of the specification alone can ascertain, at least in theory, internal completeness, which is a property of the specification itself. Mutual consent (e.g., a contract) determines external completeness.

Having a well-defined notion of completeness is important for two reasons. Most obviously, incomplete requirements will result in the wrong system being built and, ultimately, higher costs to correct the errors. Second, a notion of completeness is important to managing the development process. The developer must know how complete requirements are to measure the progress being made. Without a well-defined notion of completeness, it will not be clear how far along the project is or when the requirements phase is done.

The current work concentrates on methods for assessing and achieving internal completeness. Many of the activities associated with determining external completeness are part of the overall process of customer/developer communication that the Consortium will not fully address until it defines overall process (in 1992). Nonetheless, the two properties are related since the analysis of a specification for internal completeness will point out areas where the developer has not fully analyzed the requirements. For instance, an assessment of internal completeness might show that no values for a particular output are specified for one of the system modes. In such a case, the developer must go back to the customer to determine whether this represents an incomplete requirement or an instance where the customer does not care what the output value is. The engineer then makes the specification internally complete by specifying the required value or a "don't care" for the output, which accounts for all the possibilities. Thus, this discussion will cover some issues of external completeness where these are related to the analysis of internal completeness. It simply uses "completeness," without qualification, to refer to internal completeness, being explicit when talking about external completeness.

CoRE addresses completeness in two ways. First, it uses a standard model (the four-variable model) for organizing the requirements with well-defined parts. This allows the developer to assess a specification against the model to ensure that it accounts for all the expected parts. Second, the engineer writes behavioral requirements in terms of formal mechanisms like conditions, finite state machines, and functions. These provide a formal notion of completeness that he can directly assess in a specification.

As discussed earlier, the Consortium's approach to specification is based on describing requirements in terms of a specific set of relations on environmental variables. It is characteristic of CoRE that, once the engineer defines the environmental variables, the model itself defines what it means for a

specification in terms of those variables to be complete. For instance, the developer must fully define each of the component relations REQ, NAT, IN, and OUT. Taking REQ as an example, the underlying model determines what it means for REQ to be completely specified. Overall, the REQ relation must have a domain that includes all of the possible values of the monitored variables (otherwise, the required behavior would be undefined for some input values). The range must include every one of the controlled variables. The relation itself must associate a set of controlled variable values with each possible state of the monitored variables at any time.

Breaking the relation into a set of parts makes writing a complete definition of the relation (with confidence) practical. There is exactly one output function written for each controlled variable. Along with the function, the developer specifies the allowed tolerance in value and response time. This reduces the task of checking the completeness of the relation to that of checking the completeness of the function—a procedure that is well-understood.

The use of formal mechanisms like predicates and finite state machines supports the ability to check the controlled variable functions for completeness. The developer writes the states of the monitored variables in terms of the states of finite state machines and conditions on the environmental variables. Thus, he can appeal to the formal notion of a state machine to ensure that its definition is complete. He can also check the conditions to see that he has defined a behavior for all boolean cases (their union adds up to true). Then, if the function covers all the states and all such conditions, he knows its domain is complete. If all possible values of the controlled variable are accounted for, the function itself is complete. Similar techniques apply to the other relations.

Because the engineer knows what information must appear in each model, he can design a standard form for its representation that makes it clear whether all needed information has been recorded. Each form has particular characteristics that must be satisfied before a model expressed in that form is complete. For example, the information model includes an ERA model. Such a model is incomplete if an entity class exists without attributes. See Section 3.10 for the full list of completeness criteria.

Section 3 gives a more complete discussion of the specific techniques and method or representation supporting those techniques (e.g., the use of tables). For instance, the use of a standard model allows the developer to provide a standard view of the information in the model and standard templates for representing such information. He can easily check these for missing information.

5.3.2 Consistency in Specifications

A specification is consistent if it contains no internal contradictions. In other words, the specification is not self-contradictory or ambiguous; there is no more than one behavior required for any given situation. The method requirements dictate that CoRE will support the development of consistent specifications. "Support" in this case implies two capabilities. During requirements analysis and specification, CoRE should assist the analyst in writing consistent requirements. Second, the underlying model should have a useful definition of consistency that permits a straightforward process for verifying consistency.

CoRE applies a two-part strategy. First, it uses formal models for which the notion of consistency is well-defined. This allows the consistency of certain aspects of the specification to be demonstrated by analysis. Second, it applies a set of principles that assist the analyst in developing consistent specifications or the reader in assessing the consistency.

The ability to demonstrate consistency motivates, in part, the Consortium's choice to define the behavioral requirements as a set of relations on environmental variables using predicates, state machines, and functions to write the specifications:

- The use of predicates allows the appeal to boolean logic in developing consistent specifications or checking for consistency. For instance, predicates define different states of interest as in: "if x<y output a, if x> =y output b". Mutually exclusive predicates ensure the consistency of such specifications; i.e., if one condition is true, the others must be false. Such analyses are typically not difficult and are an exercise common to anyone who has programmed.

- The use of functions to specify the required values of controlled variables provides a formal interpretation of consistency. By definition, a function maps each value in the domain to no more than one value in the range. Thus, a well-formed function ensures the consistency of the output specification.

- Similarly, the use of deterministic finite state machines supports consistency in the definition of states and state transitions. For a deterministic state machine, a function again defines the possible state transitions. For any given state/event pair, there is only one possible next state. The developer can ensure that the state transitions specification is consistent by ensuring that the state machine is well-formed in this regard. He can do this mechanically if the definitions of events are consistent.

- These properties simplify the consistency checking of the controlled variable functions. The range of these functions is defined in terms of the states of state machines (called modes) and conditions. Since a finite state machine can be in only one state at a time, defining the function this way reduces the consistency check to ensuring that the conditions are exclusive and the same mode does not correspond to two different outputs. Section 3 discusses this in more detail.

Specifications in which information about a particular requirement may be redundant or spread out through the specification (as in typical in prose specifications) invite inconsistency. Over the course of time, requirements will change. Where information is spread through the specification, it is likely that some of the relevant parts are missing. Such specifications become increasingly inconsistent over time.

CoRE avoids such problems by applying the principles of nonredundant specification and separation of concerns. The overall organization of the data model leads to specification in which there is exactly one place to put or find a given piece of information. Further, the separation of concerns (implemented in the object organization) localizes all of the information that is likely to change together in a single object. As long as the object interface is not affected, all of the information that must change will be in the same object. This makes it easy to find what must be changed and effect changes without introducing inconsistencies.

A variety of other specific techniques are applied to assess consistency depending on the choice of methods of representation. For instance, the tabular representation of functions supports visual inspection to determine some kinds of consistency. Section 3 discusses these issues in more detail.

5.3.3 Specifying Timing and Accuracy Constraints

CoRE must manage timing and accuracy constraints for real-time systems. Developers can then evaluate timing and accuracy constraints for feasibility and can remedy impossible or very difficult

constraints early in the development process. All of the relations—REQ, NAT, IN, and OUT—deal with the behavior of physical things. All relations must take into account the variation from ideal values (error) and delay inherent in all real systems.

Tolerance can have two slightly different meanings: it describes either the behavior that exists (i.e., NAT), or the behavior that must be satisfied (i.e., REQ). Tolerance refers to the variation and delay that is acceptable for the REQ relation the software must satisfy. This report refers to all requirements constraining the allowed delay and variation as "timing and accuracy constraints."

Timing constraints express offsets in the time parameter of the variables. Those offsets define the amount of delay that is acceptable in the implementation, e.g., the allowed delay between the time a button is pressed and the corresponding outputs are produced. The simplest delay would be a fixed value. This method is also consistent with the complex behavior of delays associated with the real world, such as device behavior.

Accuracy constraints describe the expected behavior of the environment and devices, NAT, IN, and OUT. They also describe the tolerable behavior of the system, REQ. In both cases, formal descriptions of behavior introduce error terms and error functions. Expressing the controlled variables functions allows end-to-end accuracy constraints to be expressed as an allowed tolerance in the values produced by the function. Typically, this is easily expressed as an error term. The well-defined principles of error and precision come from a formalization of behavior The calculus chosen to express behavior on the variables should therefore support reasoning about error.

This method expresses behavior in terms of variables which are functions of time. This is ideal for expressing timing constraints. Since each variable is a function of time, then the engineer can introduce delay terms to accurately describe the time it takes for one variable to influence the values of other variables.

5.3.4 Tool Support

An important part of CoRE is its ability to exploit graphical editors and tools built to support other methods. This satisfies the method requirement that the Consortium method exploit existing member investment in CASE tools. Therefore, the Consortium is experimenting with the most commonly used CASE tool (**teamwork**, according to the workshop results), to show its users how to extract the full benefit from it when using CoRE. The Consortium will eventually provide more complete guidance in using CoRE with **teamwork** so it can be used in a pilot project.

However, some features of CoRE, such as its underlying formal model, offer the opportunity to automate aspects of the requirements analyst's job which have previously been unassisted. The Consortium is constructing a prototype tool intended to demonstrate such assistance. Ideally, this tool will serve both as a prototype for pilot application and as the basis for a vendor alliance that will either develop such a tool or add features to an existing tool (e.g., **teamwork**) so that CoRE will be supported directly.

5.3.4.1 The teamwork Tool

The Consortium has entered two models of the FLMS example into **teamwork**. One reflects the "ideal" structure of the FLMS specification, independently of **teamwork**'s methodological assumptions. The other tries to maximize "cooperation" from **teamwork**, in the sense of arriving at a model which is clear of offending messages, after applying standard error and syntax checks.

Both models were useful. Most of a model is readily stated within the assumptions of the CASE tool; that is, the developer can readily phrase any specification created using the method in terms of a context diagram, C-Specs, and other traditional components of Structured Analysis. Such a model shows a slight loss of fidelity, but little that customers will care about.

For instance, one casualty is the font conventions used in the textual example. The **teamwork** tool does not provide italic fonts in the presentation of its data directory, so predefined terms are not distinguishable from variable names. This aspect of the notation is of little interest to newcomers to the method. Of more concern is the mapping from mode-tabular function descriptions: **teamwork** has three forms of tables (state/event matrix, process activation table, and decision table), but none is an exact match. However, omitting the syntax check on the state/event matrix allows at least an appropriate cosmetic and the use of a generalized table editor to speed the entry process.

If the developer is willing to use the accepted conventions of a standard notation (such as the Boeing-Hatley conventions), as the Consortium does in its "cooperative" model, then he may use the useful input error detection which **teamwork** can apply based on the input conventions. In general, he can enter all parts of CoRE's data model for subsequent retrieval. At this point, the Consortium would conclude that the attempt to cooperate with **teamwork** is more profitable than the attempt to fight with it.

A key part of the Consortium's development of CoRE was attention to creating a representation-independent data structure. The reason for that attention was to ensure a reasonable fit with existing CASE tools. Based on results, that effort was worthwhile. There is no reason to suspect that either of the two paradigms for modeling with the CASE tool would preclude the use of existing extensions for document generation (e.g., for compliance with DOD-STD-2167A). The real question is, how much of the built-in error detection and syntax checking can be used, and how much must be bypassed. The Consortium is currently exploring the limits with an eye to extending them.

5.3.4.2 Ideal Requirements Toolset

A conceptual prototype of an "ideal" requirements tool to accommodate CoRE is currently under construction, on a Macintosh. It uses an underlying hypertext engine (Supercard) to support some of the critical features of the approach. The Consortium chose the Macintosh/Supercard environment because these facilities provide an excellent prototyping environment for trying and assessing tool features. Any actual tool developed through vendor alliance would run on common member company platforms.

Of the requirements for a methodology which the Requirements Workshop emphasized, the prototype most heavily focused on those which are **not** well met by available commercial products. The Consortium's intention is that the user look at its prototype in conjunction with commercial tools for lessons it teaches about what capabilities a mature CASE tool could and **should** contain.

Included novel features are:

- Requirements management facilities allow incorporation of pre-existing documentation into the model, where it becomes accessible by hypertext mechanisms. Thus, the user gets fine-grained access to details of the requirements database, while creating and editing the model are actually simplified. Additionally, the user may present the entered requirements with graphical or textual views, or both.

- Requirement objects may be reached by multiple paths. Consequently, multiple user-definable views of system structure may be created. This allows the clearest possible communication

between "producers and consumers" of the specification by allowing views designed for different sets of needs and interests. Redundancy in the specification is not materially increased.

- Dictionaries may be associated with different paths of descent through the model. Thus, definitions within an object may differ depending on the context in which the object is viewed.

- Models may be interpretable ("Executable requirements"). This is a consequence of the semantic detailing extractable from the underlying formal model. This feature can provide considerable dynamic diagnostic power, one use of which is to apply consistency checking against derived requirements.

- Use of a representation-independent data structure, so that it is possible to map to diagram types associated with presently-used methodologies (e.g., data flow diagram, state transition, ERA) from the common data model. The prototype tool demonstrates this generative power.

- The tool allows user-defined graphic symbols ("visual macros").

5.3.5 Verification and Validation of Specifications

The requirements are the primary vehicle supporting verification and validation. Verification comprises all activities directed toward ensuring that the implementation satisfies the stated requirements (does the job right). Validation is the task of ensuring that the software fulfills the intended purpose (does the right job). Different methods vary in the the amount of support the resulting requirements provides for these activities; e.g., the amount of detail provided, how easy the information is to extract from the specification, or the ability to distinguish actual system requirements from incidental behavior. The Consortium's approach, with its standard, formal model and emphasis on observable behavior, provides for a systematic approach to both verification and validation.

5.3.5.1 Validation

Because validation inherently depends on subjective activities (i.e., the customer's perception that the system described meets his expectations), there can be no strictly analytical process for validating requirements. Typically, developers rely on a variety of techniques for summarizing and presenting the expected behavior in terms the customer will understand, such as scenarios, mock-ups, and prototypes. A method supports such validation activities to the extent that it helps the developer determine the complete, externally visible behavior in a straightforward and systematic fashion.

Although there is little empirical evidence to assess CoRE's support for validation, there are some key features that directly support validation activities. First is the use of a rigorous model for describing externally visible behavior. The method explicitly identifies exactly those aspects of the external, visible environment affected by the system in the monitored and controlled variables. These variables (e.g., temperature, pressure, position, velocity, etc.) are domain-specific and typically reflect the terms in which the customer thinks about the problem. The required behavior of the system is then given as a set of piecewise-continuous functions of these variables. That is, the functions directly express the relation between observable changes in the system environment and the observable responses; e.g., the valve is closed when the temperature exceeds 500 degrees. This allows the customer or developer to concentrate on the visible behavior and explore it systematically without needing to understand a lot of extraneous detail. This facilitates the generation of accurate scenarios since the output functions make the behavior clear, particularly at points

of discontinuity. The customer can use the functions to quickly answer specific what-if questions. Appendix A gives a more detailed walk-through of these activities, using an example specification created using CoRE as part of the guided tour of the example.

CoRE also facilitates prototyping and other activities based on simulating execution of the proposed system. Since the specification is based on finite state machines and functions, the developer can step the machine through sequences of events and systematically determine the resulting behavior. The specification is not directly executable since the expected behavior is actually defined by a relation, i.e., the specification gives the tolerance or allowed range of possible outputs. For instance, the requirements say that a weapon must be released within some acceptable window, say plus or minus 5 milliseconds of the pressing of the release button, or that a radar must be pointed to the required angle plus or minus 0.01 degrees. Thus, the specification describes all acceptable implementations, not just one. Prototyping the system from the requirements requires making some decision about the specific behavior the prototype displays for each tolerance. Otherwise, CoRE describes the observable behavior in a "machine-like" format; hence, the given specification need only be extended with such decisions about the specific behavior.

5.3.5.2 Verification

All verification activities ultimately depend on associating a set of inputs with a set of outputs and assessing the deviation from expected values. Testing remains the primary verification activity, although the tester may use other activities, from walk-throughs to formal analysis. A specification is verifiable to the extent that it facilitates such activities. In particular, it is important that the specification resulting from CoRE serve as the "test-to" specification, allowing the developer to systematically determine exactly the permissible range of outputs for a given input or input sequence.

Because verification is one of the most important downstream uses of the requirements, CoRE focuses on producing a "test-to" specification. As discussed in Section 4, the REQ relation gives the externally visible behavior in terms of the environmental variables monitored or controlled by the system. What the developer additionally needs to make the specification testable is the precise specification of, on one side, the system inputs and their relation to the monitored variables and, on the other, the system outputs and their relation to the controlled variables. The In and OUT relations specify these completely and explicitly. The IN relation describes exactly what the system inputs are, down to the level of the command sequence for reading the registers and the unconverted bit strings received as necessary (although the developer may also use a more abstract representation of input values if he does not yet know or need of this level of detail). The IN relation defines how the input value corresponds to a given monitored variable, e.g., which variable is being measured and with what accuracy and precision. The OUT relation provides similar information on the output side. This allows the tester to systematically extract all the values needed for testing. The tester can derive the observable behavior, allowed tolerances, and points of discontinuity (e.g., maximum and minimum values in the input range) from the REQ relations. He can then ascertain the exact inputs and outputs corresponding to these values of the monitored variables from the corresponding IN and OUT relations. He can thus systematically derive a complete set of test values and corresponding outputs with their acceptable accuracy from the document. This systematic and rigorous approach offers the possibility of automated test case generation and links to formal verification methods in the future.

5.3.6 Maintainability and Ease of Change

For many projects, the single greatest source of difficulty is changing requirements. This includes changes during requirements analysis whether due to better understanding of the problem or the whims of the customer, as well as changes originating in subsequent development phases whether for system evolution, maintenance, or debugging. Regardless of the source, apparently minor changes in requirements can potentially ripple not only through many parts of the requirements specification, but throughout the products of all subsequent phases of development with increasingly deleterious effect. Thus, managing such changes and limiting the effects is a key goal of CoRE.

CoRE addresses the problem of changing requirements in five basic ways:

- ***Process.*** First, it develops an analysis process that gives forethought to which aspects of the requirements are likely to change, and which are not, both in the short and long term. This includes the use of domain or problem analysis in the early development phases (e.g., Ward's CASE Real-Time Method for domain analysis), with explicit attention to capturing knowledge about what is stable and what is volatile in the problem domain. These activities are outside the scope of this report but will be addressed in subsequent detailed descriptions of the overall process. (Ward 1989) provides a summary overview of some applicable techniques.

- ***The Semantic Model.*** The semantic model focuses on nonalgorithmic specification of the externally visible behavior in terms of environmental variables. The environmental variables of interest in a particular problem domain are generally more constant from one instance to the next than subsequent (and dependent) decisions such as the choice of particular algorithms. A given requirement specification in terms of environmental variables corresponds to many possible designs and implementations and is correspondingly more stable for certain classes of changes than a specification that relates inputs to outputs or expresses requirements in algorithmic terms.

- ***Object Oriented Data Model.*** The Consortium chooses to specify requirements in terms of objects. Objects provide a mechanism for encapsulating information likely to change and abstracting from details. Just as objects to hide design decisions that are likely to change in the system design, they can hide requirements details that are likely to change. The problem analysis process explicitly identifies which requirements are stable and which are probably not, as well as which requirements are likely to change together (e.g., all the attributes associated with a target). The specification phase then encapsulates as many of the details that are likely to change as possible inside objects. The object interfaces define aspects of the encapsulated requirements that are not likely to change.

  This approach differs from many of the current object-oriented analysis methods in that there is no requirement that the objects correspond to distinct physical entities in the real world. In this approach, the engineer chooses (creates) explicitly to hide information likely to change and abstract from irrelevant detail. Clearly, objects chosen by these criteria will often correspond to physical objects since physical objects tend to change as units, but this is not always the case. For instance, the same hardware device may perform two or more tasks that are related only by the physical platform. Such a device might correspond to two or more objects.

- ***Mutable Representations.*** At the more detailed level, CoRE uses a set of techniques for specifying requirements that make it easy to locate where particular changes must be made

and allow many kinds of changes to be made while affecting only a small part of the specification. For instance, the use of state machines and function tables (treating tolerance separately) simplifies many kinds of changes. Where the details of how the current state is determined must change, these changes will be restricted to the state machine definitions. Organizing the functions as tables allows changes to the functions to be made row by row. Subsequent sections discuss these issues in more detail.

- *Automation.* While the developer can apply CoRE with little or no automation, the Consortium explicitly designed it to benefit from tool support. The common underlying data model and rigorous semantics provide a basis for tool-supported impact analysis, tracking, and dissemination of changes. Such features are currently being developed in the tool prototype.

While no method can hope to deal effectively with every sort of unanticipated change, CoRE offers a variety of methods to address those classes of changes that the Consortium can anticipate (while providing a general robustness against unanticipated changes). This support runs from the early phases of development in the modeling of the problem and analysis of likely changes, through the requirements organization as information hiding objects, through the detailed specification techniques.

5.3.7 Understandability and Communication of Specifications

The requirements specification necessarily serves a variety of purposes for a variety of users. It often serves as the primary vehicle of communication between systems and software engineers. It is the design-to specification for implementors, the test-to specification for testers, the primary vehicle for communicating between the developer and customer, and ultimately the basis of their contract. Ideally, the requirements specification should serve the needs of all these users equally. However, the ideal is difficult to achieve since there is no generally agreed upon common language shared by all the interested parties. Furthermore, an organization of the information suitable for one purpose (e.g., as a reference) will not be conducive to another (e.g., an overview of the system).

Though no single organization or presentation of requirements can hope to serve such a diverse community of users equally, CoRE can do much to address the problem through a combination of strategies.

- *Use of Graphics and Text.* By providing equivalent semantics in both a graphic and textual form, the same information can serve more than one purpose. The graphic (information and data model views) presentation is useful during the more informal, initial phases of development and provides a vehicle for quickly communicating relationships to systems components to systems and software staff. It also provides an easily learned language for communicating with the customer. Even after the specification is complete, the graphic interface provides the quickest way to get a grasp of the overall requirements organization and system purpose.

  Because there is a textual equivalent, the same specification will also serve the needs of developers, testers, and maintainers. Detailed requirements that cannot easily be expressed graphically are given in textual form. This provides the reference capabilities needed by developers and maintainers.

- *Four Variable Model.* Division of requirements into a set of relations based on environmental variables helps separate concerns, allowing the needs of different users to be served. Generally

the customer is interested in the visible effects of the system (what is it going to do for me?) and not in the details of implementation (which inputs are used for what). This is exactly the information that the REQ relation conveys. That is, the monitored and controlled variables represent visible quantities with which the customer (or systems engineer) is familiar. Since the REQ relation defines all the the visible behavior exclusively in terms of the monitored and controlled variables, the customer (or others) can understand this part of the specification without having to read a lot of extraneous detail (e.g., the IN and OUT relations). While more formal and complete than scenarios can be, the REQ specification is written in much the same terms.

- ***Relations.*** The detailed specifications are based on relations, functions, truth conditions, and state machines. These have the advantage of both having strong formal bases and being familiar to both hardware and software engineers through programming and basic math courses. Such specifications also have the desirable property that they are easy to learn to read even for those without the training to produce them.

- ***Common Data Model.*** Rather than offer a particular documentation format, CoRE is based on a common schema for requirements information. Different projections of the underlying data create different views of the requirements. This drives the equivalent graphic and textual specification. The mechanism is intended to allow other views as well, so the views adapt to the information needs of a particular class of users. One important case of this is the generation of documentation conforming to MIL-STD-2167A.

  A practical exploitation of multiple views requires tool support. This is one area where CoRE has been developed with forethought about how automation will help address some problems that paper documents cannot easily address. The Consortium's work in tool prototyping is currently exploring issues in providing customized views of a requirements database created using the common data model.

5.3.8 Redundancy in Requirements

The use of redundancy in a requirements specification brings two goals into conflict. Redundant information can support understanding and ease of use. For instance, it is useful to provide graphic summaries of information that is given in detail textually. On the other hand, redundancy can lead to inconsistencies. Where the same information appears in two or more places, the developer must ensure that the information is consistent everywhere it appears. In many cases, a change made in one place will not be reflected everywhere the information is redundant, so the specification becomes increasingly inconsistent as the developer makes changes. This is particularly a problem where he uses prose in paper documentation and there is not a consistent, rigorous model for the specification semantics. The overhead of keeping such documentation consistent is more than most projects can easily afford.

The Consortium attempts to get the benefits of redundancy with few of the drawbacks by using a common, rigorous underlying model for the information structure and semantics of the requirements. CoRE focuses on specifying the requirements information in a common, underlying information model called the requirements data model. Different representations of the requirements must be directly related to this underlying model so that the transformation from one representation to another is well-defined. In particular, CoRE defines the graphic representation and the textual representation so that there are exactly equivalent representations in either where the semantics overlap. Thus, there is a

procedure for turning a graphic into a textual specification and vice versa. This allows the developer to use the graphic form in sketching out ideas and exploring alternatives for the early phases of development, or later to provide overviews of the system as is done in the example specification provided. He can then directly translate the graphic view to a textual specification.

Fully exploiting these capabilities will require tool support. Where automation is available, the developer can use data model to drive different views of the requirements, and he can automatically reflect changes in view in the other views. The Consortium's work in tool prototyping is exploring methods for implementing such capabilities.

5.3.9 Feasibility of Requirements

The Consortium's approach to requirements differs from some other state-machine-based methods, and particularly the work on "executable specifications" (e.g., [Zave and Schell 1986]) in that it focuses exclusively on specifying required behavior. It treats the issue of whether the behavior required is, in fact, feasible, as a distinct issue.

A set of requirements is feasible if there is at least one implementation that satisfies them. Clearly it is important that the developer has feasible requirements if he is to develop any actual system. Thus, demonstrating that requirements are feasible is a major part of risk reduction activities. Particularly where the system must do new kinds of tasks, meet stringent timing constraints, or meet unprecedented levels of accuracy, it is important to ensure that some implementation satisfying the constraints actually exists.

A variety of techniques can assess feasibility, including rapid prototyping and the use of executable specifications. While the activity is necessary, its goals inherently conflict with other goals of the requirements phase. In particular, it is generally in both the developer's and customer's best interest to capture only the actual requirements, leaving the implementor as much freedom as possible to choose the best implementation. Thus, a major focus in developing requirements methods has been directed toward describing what the software does, leaving the description of "how" to the design and coding phases. Since feasibility demonstrations inherently require that the "how" be specified, methods that focus on producing feasible specifications tend to unnecessarily constrain subsequent development phases. Where these concerns are mixed together in a specification, the reader cannot tell which part of the specification represents the actual system requirements and which part represents arbitrary decisions made to support feasibility. The specification becomes substantially larger and unnecessarily difficult to understand, use, or verify.

For these reasons, the Consortium's approach treats feasibility demonstration as an issue distinct from requirements specification. In particular, the system values are given as relations (expressing tolerance) rather than as functions, and the outputs are expressed in terms of monitored variables, not the system inputs. Thus, the specification is not fully executable. To make the specification executable requires adding information about how specific values are derived (i.e., how an aircraft's position is calculated from accelerations of an inertial platform), and which values in the range of allowed tolerance are produced. This should be a simple extension of the current specification, but CoRE has not yet directly addressed this. However, such demonstrations are a part of the CASE Real-Time Method, so applicable methods for creating the necessary extensions exist.

This page intentionally left blank.

6. CONCLUSIONS

To the extent that the Consortium has been able to validate CoRE with small examples, the approach appears sound. The Consortium has achieved the initial technical goal of merging a strong, graphic-based method with a formal, text-based method with the desired result of being able to exploit the best features of each approach. The Consortium has successfully applied CoRE to a small, real-time problem, with results that are consistent with the high-priority method requirements (see Section 5). In particular:

- CoRE provides suitable mechanisms for defining behavioral, timing, and accuracy requirements for real-time embedded systems.
- CoRE provides a variety of mechanisms to accommodate changing or fuzzy requirements.
- CoRE supports communication by allowing specification in problem-specific terminology, behavioral modeling in terms of the problem domain, and the use of graphics.
- There is an underlying common data model that is document- and standard-independent but appears mappable to DOD-STD-2167A.
- The object-oriented approach supports separation of concerns, localization, and concurrent development.
- The underlying, standardized formal model supports nonalgorithmic specification and systematic checking for completeness and consistency.

Since this is an interim report, a variety of needs remain to be addressed. These include specific mechanisms to handle traceability, the extension to specification of command and control, and guidelines for mapping CoRE and its products to specific standards. Of particular concern are issues of scale and integration with the system requirements process. The work to date confirms the need to address at least some aspects of the system requirements problem to fully address open issues in software requirements.

Work in the next development cycle will concentrate on the remaining member company needs. Work in the remainder of this year will focus on completing the first cycle of tool prototyping and on arranging a suitable pilot application of the method in an IR&D effort. The pilot application and supporting method development in 1992 will focus the remaining method requirements and provide more realistic examples. The Consortium will address all of the critical issues necessary to a practicable method by the end of 1992. It has scheduled a complete guidebook on the method for 1993.

This page intentionally left blank.

APPENDIX A. FUEL-LEVEL MONITORING SYSTEM INTRODUCTION AND GUIDED TOUR

A.1 INTRODUCTION TO THE FUEL-LEVEL MONITORING SYSTEM EXAMPLE

As a vehicle for exploring and validating the method during development, the Consortium applied it to a small, real-time example. The result is a small requirements specification, the FLMS Specification, in Appendix B. This example illustrates many of the features of CoRE including the use of both graphic and textual specification, the organization by object, and the use of formal models. Section 3 discusses the methods applied to develop the specification. Section 4 discusses the notations and presentation techniques used. This appendix provides a walk-through of the FLMS specification from the points of view of different users. The walk-through serves as an introduction to using the specification and illustrates how the specification would be read to answer the kinds of questions different users have.

While the example illustrates many of the issues addressed by CoRE, there are a few that are not. In particular, the example does not deal with scale-up issues. Scale-up issues will be addressed in future reports. It also does not address:

- Nondiscrete feedback: All of the controlled environmental variables are discrete.
- Creation and destruction of entities: All of the entities in the example are unconditionally related to each other, which fails to deal with real-world examples such as target acquisition.
- One-to-many relations: There are no instances in the example illustrating interesting object relationships that are not one-to-one.

A.2 PROBLEM DESCRIPTION

The problem is a simplified version of a safety shutdown system that is part of a shipboard fuel-level monitoring and control system. The overall system provides fuel to the engines and moves fuel between the shipboard tanks to ensure a constant supply and help maintain trim. The safety shutdown system is a separate component of the overall system that shuts down the fuel pumps under unsafe conditions such as too low or too high a fuel level in a tank. The Consortium simplified the problem by allocating a single tank and a pair of pumps to the software component specified (a similar system would monitor the other tank/engine pairs). It also assumed a relatively simple method of measuring the fuel level based on differential pressure in the tank; i.e., it did not address issues like extreme roll or pitch that would complicate the measure of fuel level in a real shipboard system. The FLMS problem is based on a similar problem (van Schouwen 1990). The prose description of the problem is as follows:

> The design of a fuel control system typically comprises automatic and/or manual control mechanisms (engine and fuel-level control) and safety monitoring devices. The safety monitoring devices include:

fuel gauges and gauge cocks that convey the fuel level in the tank; fusible plugs or fuse alarms that alert the operator when the fuel level is too low; fuel flow rate gauges and other gauges showing the engine operating conditions. The FLMS is intended to replace and/or complement the above-mentioned devices. It monitors and displays the fuel level in the tank, and provides visible and audible alarms for high and low fuel levels. With the currently selected hardware configuration: fuel level is displayed in a window on a CRT display; two "annunciation" windows on the CRT provide visible indication of exceeded fuel-level limits; and the computer's speaker provides the audible alarm.

In addition to annunciation windows and the alarm, the pumps are shut down under the following conditions: (1) when the fuel level is too high, since an overly high fuel level can cause fuel-hammer, leading to pipeline rupture; (2) when the fuel level is too low, since an overly low fuel level may result in the engine running dry and being damaged; and (3) when the monitoring system itself fails as indicated by its failure to reset a watchdog time-out mechanism. It is assumed that the shut-down mechanism is relay operated. Hence, the FLMS outputs a single signal when the pumps are to be shut down.

The FLMS provides two push-buttons that are used for the following purposes: (1) the button labelled, SELF TEST, allows the operator to check the FLMS's output hardware while the system is shut down; and (2) the button labelled, RESET, allows the system to be brought nto normal operation, following a shut-down or testing, as long as the fuel level is within a specified range (van Schouwen 1990).

The Consortium's specification covers only the software (not system) behavior. One consequence is that the Consortium does not directly address some issues of hardware failure (thus, van Shouwen's mode class Failure and its submodes are not present in the FLMS problem). The behavior of the FLMS is not specified for system failure.

A.3 A GUIDED TOUR OF THE EXAMPLE

The method requirements state that the specifications developed by CoRE must be suitable for a variety of users. The requirements serve as the basis for interacting with the customer, they provide the design-to specification for the software developers, and they are the test-to specification for those verifying the implementation. Each such group approaches the specification with a slightly different set of questions that the specification must answer. This guided tour illustrates how different users might exploit the organization of the specification to answer their questions. In doing so, it also illustrates how the specifications should to be read.

The software requirements specification of the FLMS is organized in a form that facilitates its use as a reference. However, it can be also used to obtain an overview of and an introduction to the software requirements. This specification assumes there is also a system specification, so it does not provide a complete introduction to the domain in which software is to be applied. It is organized as a reference document on the software requirements for software designers, implementors, and testers who need precise answers to specific questions. It is the definitive set of software requirements and will be used to validate the software.

To demonstrate how it can be used effectively by different audiences, this section contains three guided tours:

- Section A.3.1 is a tour for readers for whom this is their first exposure to CoRE and who wish to obtain an overall understanding of the FLMS and of the Consortium's approach to requirements specification.
- Section A.3.2 is an example of how a software designer could quickly reference the specification to obtain information (i.e., answer specific requirements questions).

- Section A.3.3 is an example of how a tester performing requirements validation might use the specification.

A.3.1 Tour for Overall Understanding

The suggested reading sequence should take no more than 30 minutes and will provide familiarization with both the method notation and the document organization. The tour is presented as a sequence of steps. The discussion does not include the system-level views that are expected to be available for reference. These would provide more context information than is given in a software specification. For instance, the reader would look first at a system-level information view such as that given in Figure 8 (page 44) to understand how the software entities relate to the rest of the system before examining the software specification itself. The following discussion will concentrate on the understanding of the software specification:

- Step 1: Read the system purpose (Section B.1.1, page 81) and study the figures. This gives an informal, prose overview of what the software is to accomplish.

- Step 2: Study the context diagram, Figure 15 (page 83). This diagram illustrates the external interfaces of the software and the entities in its environment (i.e., the system context) with which the software interacts in terms of the environmental variables. It graphically portrays the environmental variables monitored or controlled by the system, of which the software is a part. The environmental variables represent physical quantities in the real world with which the system interacts. The diagram shows monitored variables as arrows to the FLMS and controlled variables as arrows leaving the FLMS. The software specification will be written in terms of these environmental variables.

  Read the definitions of the monitored variables. These describe the information used by the system. Find these definitions by looking up the variable names in the index for monitored state variables (Section B.11.1). The bold index entry gives the page where the variable is defined.

  The variable definition provides all the information needed to understand how the variable is related to the environmental quantity the software needs information about and how that quantity is represented. The variable definition gives its type and its physical interpretation, i.e., exactly how the variable corresponds to an environmental quantity. For instance, it gives the definition of FuelLevel in terms of a length corresponding to the level of the fuel in the tank along the vertical axis at a certain point (assuming the tank is level and still). The corresponding NAT relation will give any constraints on the variable behavior, e.g., the maximum rate of change of the fuel level in the tank. This is usually more detail than is needed in a first reading.

  Similarly, look up each of the controlled variables of interest using the index for controlled state variables (Section B.11.2). These definitions describe exactly what the software affects in the environment.

- Step 3: Study the software information view. The information view (Figure 16, page 84) gives the overall organization of the requirements. The information view presents the static view of the requirements data captured in the specification.

  Look at the objects in the information view and their relations. Each object corresponds to a distinct subsection of the requirements specification. The information view presents global

information about how these objects depend on one another. The numbers on the relation lines show the required relative cardinality, in this case, one-to-one for all of the objects. More complex problems might require multiple instances of an object for each instance of another (pistons to engine block) or a varying number (threats to aircraft). In the case of the FLMS, there is only one tank to be monitored, but if there were to be multiple tanks, the line from monitors to the Fuel in Tank Interface would be annotated 1:5 if five tanks were to be controlled. The information view also shows class structuring (the "is a" relation) as is illustrated in Section 4.

Each block in the diagram corresponding to an object shows the object attributes. The attributes represent the public information available on the object interface, i.e., requirements defined in the object that can be used in the definitions of other objects. The attribute definitions can be read now or after Step 4. Read the attribute definitions to get a top-level view of the requirements in terms of the abstracted values provided by each object. These definitions may be found by looking up each attribute name in the index for interface terms (Section B.11.4).

- Step 4: Study the Transformation View, Figure 17 (page 85). The transformation view presents an expansion of the software bubble (labeled FLMS) in the system context diagram. It represents the dynamic structure of the requirements by showing what interface information is used by which objects in the system. In particular, it gives an overview of how the state machines (e.g., in the InOperation control bubble) drives the controlled variable functions. It also shows the requirements objects which determine the values of the controlled variables. For instance, since the value of each controlled variable is set by a single function, the definition of the Operator Interface object should contain four functions.

The InOperation object represents the In Operation mode class. An arrow from it to an object indicates that the value of the functions included in the object definition depends on the mode class. In this example all of the functions depend on the single mode class, although in more complex problems some functions will depend on more than one mode class. An arrow without a source to an object indicates that one or more functions included in the object definition depend on the value of the monitored variable that labels the arrow.

- Step 5: Scan the object definitions to understand what requirements information is captured in the object definitions (or some subset of interest). Find the object definitions by looking up each object, by name, in the index. Each object description has three major sections:

  - ***Interface.*** The interface section provides information about the object that can be used in other parts of the requirements specification. It specifies precisely what the writers of other object definitions can assume will not tend to change about the object. The interface sections provide information in the following categories: interface terms, monitored variables, events, and modes. The interface section does not contain controlled variables, since the information about how the system determines and sets a controlled variables value is always local to one object description.

  - ***Encapsulated Information.*** The engineer can change encapsulated information about an object without affecting other sections of the requirements specification. By carefully deciding what information should appear in the interface section versus the encapsulated information, the engineer makes it feasible for users to quickly determine the impact of requirements changes.

The encapsulated information includes local terms and controlled variables output functions. The controlled variables functions, in particular, should be scanned to understand what outputs the system must to produce in response to environmental changes. This is the heart of the REQ specification.

- ***Input and output.*** The input and output section is also part of the encapsulated information. It describes the input and output data items in the IN and OUT relations. Objects encapsulate this information since the engineer writes the remainder of the specification only in terms of the information on the object interfaces (monitored and controlled variables, and terms written as functions of these variables).

For example, examine the Pump Interface object which defines the function that sets the values of Shutdown and disables PumpSwitch. The definition of the function must give the value of PumpSwitch under all possible conditions. The controlled variable definition specifies that PumpSwitch has only two enumerated values, open and closed. Similarly, Shutdown is either true or false. The NAT relations shows that the hardware constrains the current state of PumpSwitch depending on the values of Shutdown, WDTimer, and ResetSwitch. In other words, the software does not control the value of PumpSwitch directly, but enables it or disables it by controlling the value of Shutdown (in this object) or the value of WDTimer (the Watchdog Interface object). In particular, the table specifies that PumpSwitch can only be closed (so the pump is running) if the reset switch is not currently pressed, the watchdog timer has not timed out, and Shutdown has the value false.

Section B.6.2.4 gives the actual software requirements. Here the function table shows what values that the variable Shutdown must have in each of the operating modes. For instance, Shutdown must be true (hence the pump will be disabled) in either Standby or Test mode. The tolerance specification shows that a delay of no more than 50 milliseconds is permitted in the implementation.

Finally,the output data items specify the hardware resources available. This specification shows what value must be assigned to the output register to change the value of Shutdown and, consequently, the pump switch.

- Step 6: Review the operating modes diagram, Figure 18 (page 86). The behavior of the software is specified in terms of the operating modes, which are: Operating, Shutdown, Test, and Standby. The events that cause a transition between modes are as encapsulated information in the InOperation object.

To learn the basics of the notation for describing events, step through the definition of the event that causes a transition from the Standby mode to the Operating mode:

$$InOp2 = @T(Reset) \text{ WHEN } [InsideHysRange]$$

Since *InsideHysRange* is an attribute of the Fuel In Tank Interface object, its definition is part of the description of the interface of this object. Alternatively the index for the interface terms (Section B.11.4) could have been examined. This term is a BOOLEAN that indicates whether the fuel level is between the high and low limits. The transition to the Operating mode requires the fuel level to be within these limits.

To determine the meaning of the term *Reset,* look in the definition of the Operator Interface object, of which it is an attribute. The object interface specification, *Reset,* is defined as an event. To analyze the boolean condition DURATION(ResetSwitch = pressed)> = *ResetDetected,* note the attribute definition of ResetSwitch. ResetSwitch has the pressed value when the push button labeled RESET is held down; otherwise, it has a value of released. The boolean condition is true when RESET is pushed for more than the time period denoted by *ResetDetected;* i.e., the event occurs at the end of any interval in which RESET has been pushed and held down for three seconds. The event corresponding to a system reset is public information. The details of how a reset is actually detected are encapsulated by the object and can change without affecting the specification of other objects. In summary, the transition from the Standby mode to the Operating mode occurs if RESET is pushed for a specified time period when the fuel level is between the high and low limits.

Work through the other event definitions since the modes of operation, and the conditions under which the mode changes, are important to understanding the required behavior of the software. The operating modes in Figure 18 (page 86) comprise the mode class InOperation. Many problems will have multiple mode classes to represent complex state behavior, although this problem has only one.

A.3.2 Tour for the Software Designer

The specification is organized to facilitate rapid reference for designers seeking to answer specific questions about required behavior. Assume the function that sets the value of the controlled variable PumpSwitch is to be designed. Some of the questions that might be asked are:

How is the value of PumpSwitch controlled?

How are the relevant modes determined for the portion of software being designed?

How is the allowable timing for setting PumpSwitch determined?

- Step 1. Get an overview of the required behavior of the software relative to PumpSwitch through the graphic system overviews showing which object is associated with the controlled variable, or through the index for controlled state variables (Section B.11.2). Either will identify the definition of the PumpSwitch Interface object.

- Step 2. Study the definitions of the controlled variables to understand what environmental quantity the variable models. The controlled variable definitions define the PumpSwitch variable and describe its relation to the physical switch. The definition gives the value of PumpSwitch under all possible conditions. The controlled variable definition specifies that PumpSwitch has only two enumerated values, open and closed. Open corresponds to the pumps being shut down.

  Once the physical interpretation of the variable is clear, study the NAT relation to understand the physical constraints on the variable imposed by the environment. In this case, the NAT relation shows that the software does not control the value of PumpSwitch directly, but constrains its changes in value through the variable Shutdown. In particular, the NAT relation shows that the hardware constrains the current state of PumpSwitch depending on the values of Shutdown, WDTimer, and the ResetSwitch. The NAT table specifies that PumpSwitch can

only be closed (so the pump is running) if the RESET is not currently pressed, the watchdog system has not timed out, and Shutdown has the value false.

- Step 3. Read Section B.6.2.4 to understand the externally visible behavior that the software must implement. Here the controlled variable function is given in the form of a table relating the current modes and other state information (if any) to the required value of Shutdown. For instance, Shutdown must be true (hence the pump will be disabled) in either Standby or Test mode.

The function table is supplemented with information giving the allowed tolerance in the accuracy and timing of the implementation. For instance, the tolerance shows that a delay of no more than 50 milliseconds is permitted in the running code from the change in mode to the setting of Shutdown. Similarly, a nondiscrete output would also show a required level of accuracy.

- Step 4. Use the mode names and terms in the function table to find exactly what conditions result in which behaviors (i.e., what conditions the implementation must test for). The left-hand column of the table gives all the relevant modes. These can be used to look up any needed information about the modes (through the index for modes [Section B.11.3]). Where the output is a function of additional conditions, these will either be locally defined, or imported from another object. Where they are imported, the corresponding term can be looked up in the Interface Terms and traced to the defining object. The defining object will specify the meaning of the condition, give the definition of the monitored variables that determine the value of the condition, and specify how those values are derived from the hardware input devices. Such traces can be carried into the design decomposition as well.

- Step 5. Finally, to understand exactly how the software sets the value of Shutdown, find the OUT relation on Shutdown. It shows the device behavior corresponding to software outputs. For instance, the assignment of the value "operate" (1 on bit one of Port C) sets Shutdown to false and "shutdown" (0 on bit one of Port C) sets Shutdown to true. Any loss of precision between the device and controlled variable is also shown here so the implementor can verify that the tolerance requirements are satisfied.

These steps provide all the information relevant to setting the PumpSwitch value while also giving the software designer a detailed understanding of how the software interacts with the hardware to manipulate the actual switch. Irrelevant portions of the document do not have to be studied; all of the required information is quickly available through the indexes.

A.3.3 Tour for the Software Tester

The software tester will use much of the same information as the software designer but will want to access it in a different order. This tour assumes the reader has already done the previous two tours, and material presented in them is summarized here.

Assume that the job is to test the code that manipulates the pump switch. Test scenarios and test data need to be generated:

- Step 1. Review the definition of PumpSwitch and Shutdown, the NAT relations, and the Required Behavior section (B.6.2.4) of the Pump Interface Object. This shows how the value of Shutdown affects the value of PumpSwitch and how the modes of operation, in turn, affect the value of Shutdown. Scenarios can be constructed showing that the value of Shutdown (in

a valid implementation) will change on transitions from modes Operating or Shutdown to modes Standby or Test and vice versa. Corresponding changes will occur in the PumpSwitch as specified by the NAT relation. The scenarios suggest a sequence of tests at the points of discontinuity corresponding to the mode transitions.

- Step 2. Find the definition of the InOperation mode class (through the index). This shows the operating modes and gives the conditions under which transitions occur between the modes. Since PumpSwitch is not included in the interface of the Pump Interface object, it cannot change the mode.

- Step 3. Review the OUT relation to determine the virtual events related to PumpSwitch for which tests must be prepared. This shows exactly what values must be generated (to the hardware or equivalent stubs) during unit and other testing corresponding to the values of the controlled variables.

APPENDIX B. FUEL-LEVEL MONITORING SYSTEM SPECIFICATION

B.1 INTRODUCTION

B.1.1 Purpose of the Fuel-Level Monitoring System

The purpose of the FLMS is to:

1. Stop the pumps that pump fuel into and out of the tank when the fuel level is either too high or too low.
2. Provide information about fuel level.
3. Regularly notify the Watchdog system of nominal behavior.

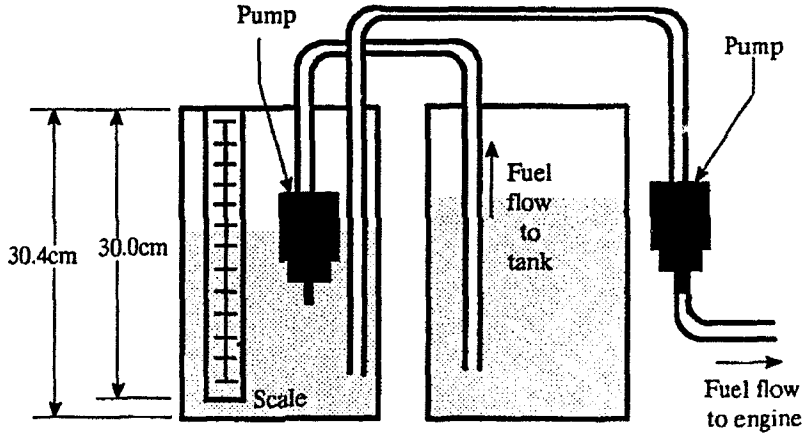

Figure 14. Fuel-Level Monitoring System: Pump and Tank Configuration (Front View)

B.1.2 Notation

This specification adopts the following notational conventions:

- **VariableName.** It uses initial capitals with interspersed capitals and lower case to indicate variable names. The variables may be monitored or controlled variables or input or output data items. Examples are FuelLevel and CursorRow.
- ***TechnicalTerm.*** It uses italicized initial capitals with interspersed capitals and lower case to indicate technical terms. Examples are *LowerCalibrationBound* and *Offset.*
- **value.** It uses all lower case to represent the values of enumerated variables. Examples are on and off.
- **METHODTERM.** It uses all capitals to represent method-defined terms. Examples are INMODE and DURATION.

B.2 SYSTEM CONTEXT DIAGRAM

Figure 15 illustrates the context of the FLMS. The bubble represents the FLMS system. Inputs to the FLMS are the monitored variables. Outputs are the controlled variables.

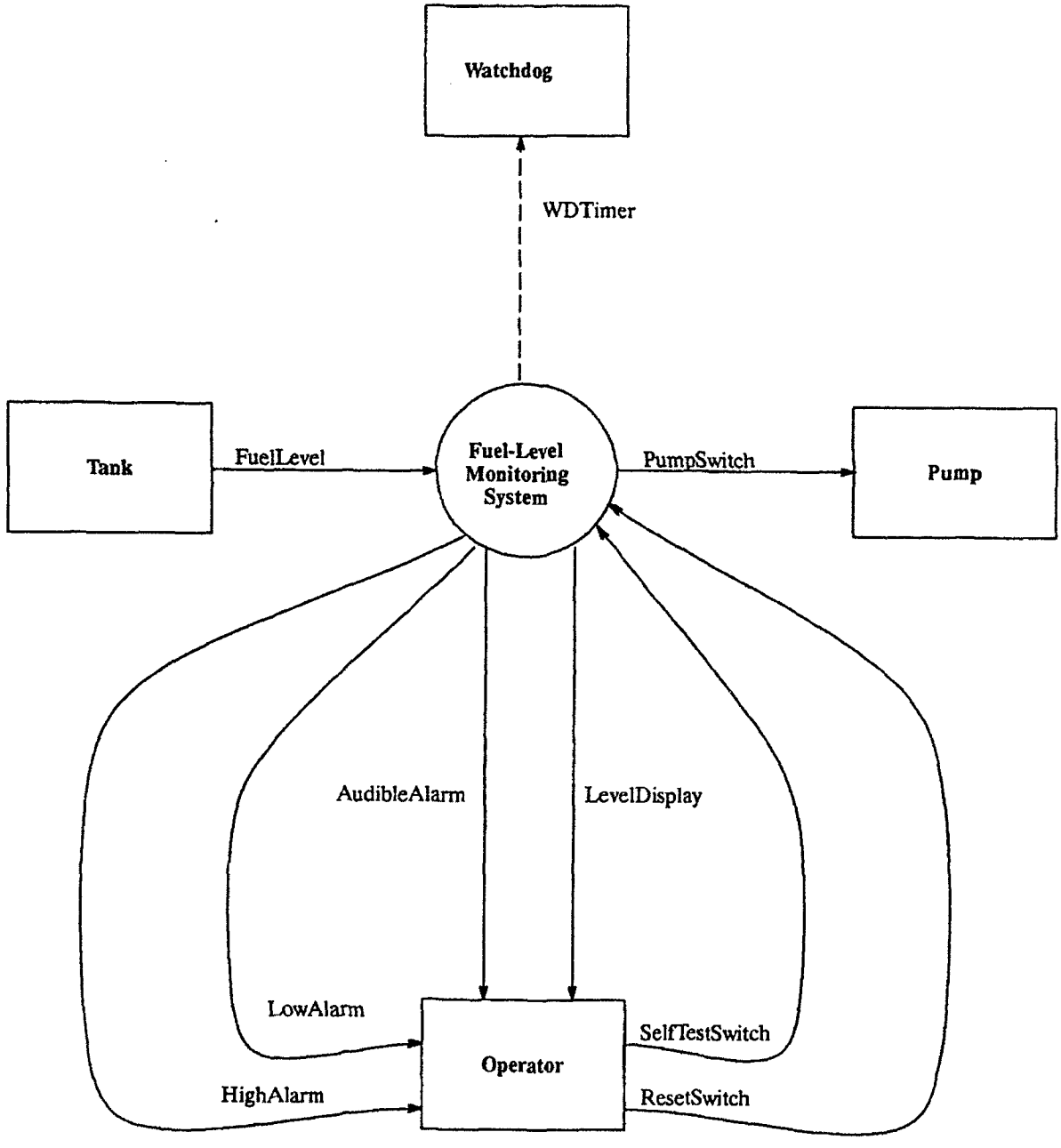

Figure 15. Fuel-Level Monitoring System: Context Diagram

B.3 FUEL-LEVEL MONITORING SYSTEM INFORMATION VIEW

Figure 16 illustrates the information view of the software. The boxes denote objects. The attributes are requirements information available to other object definitions in the requirements. Cardinality of the relations is shown on the relation links.

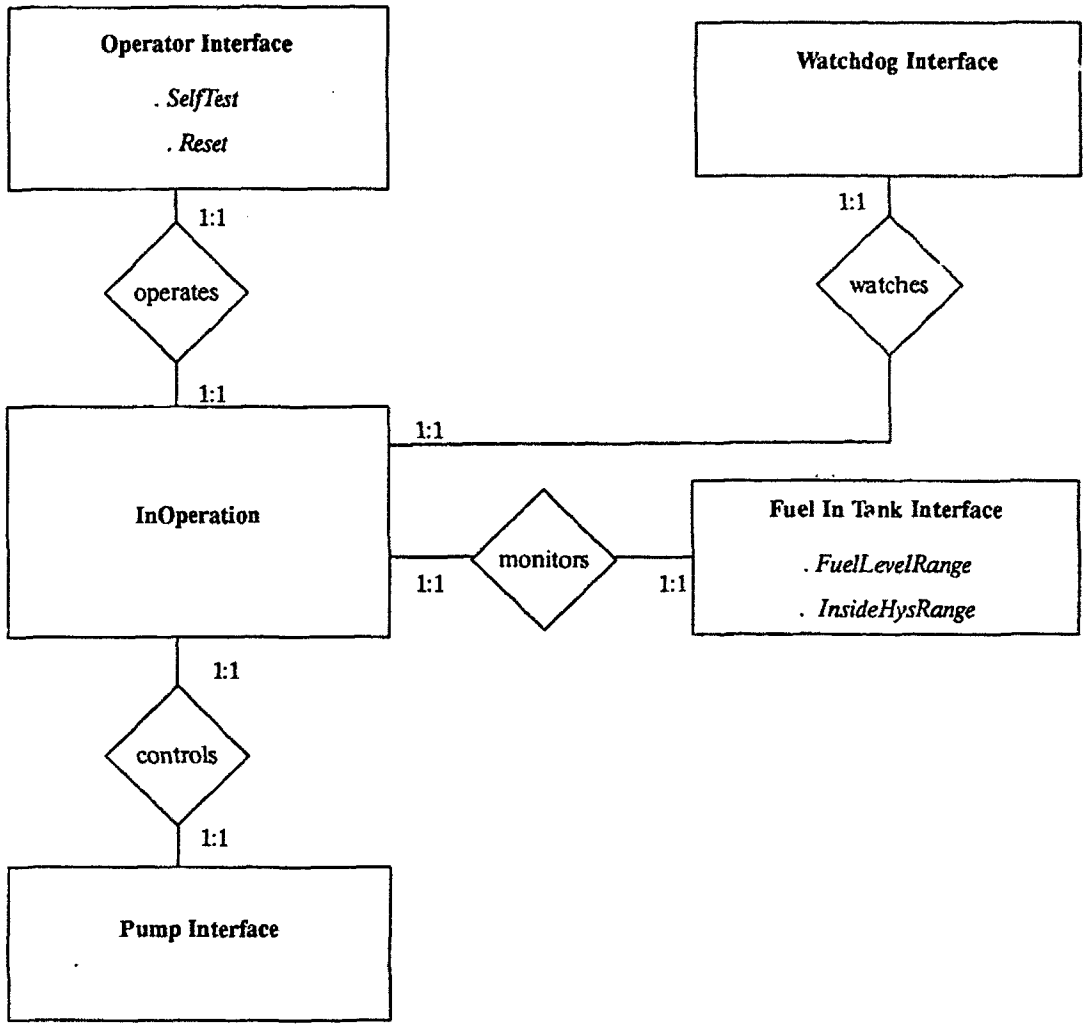

Figure 16. Fuel-Level Monitoring System: Information View

B.4 FUEL-LEVEL MONITORING SYSTEM TRANSFORMATION VIEW

Figure 17 illustrates the transformation view of the software. It is the detail view of the FLMS bubble in Figure 15. Arcs entering the diagram from outside represent monitored variables, and arcs leaving the diagram represent the controlled variables. There is an internal arc where an object uses requirements information defined on the interface of another object. These show up in the information view (Figure 16) as object attributes.

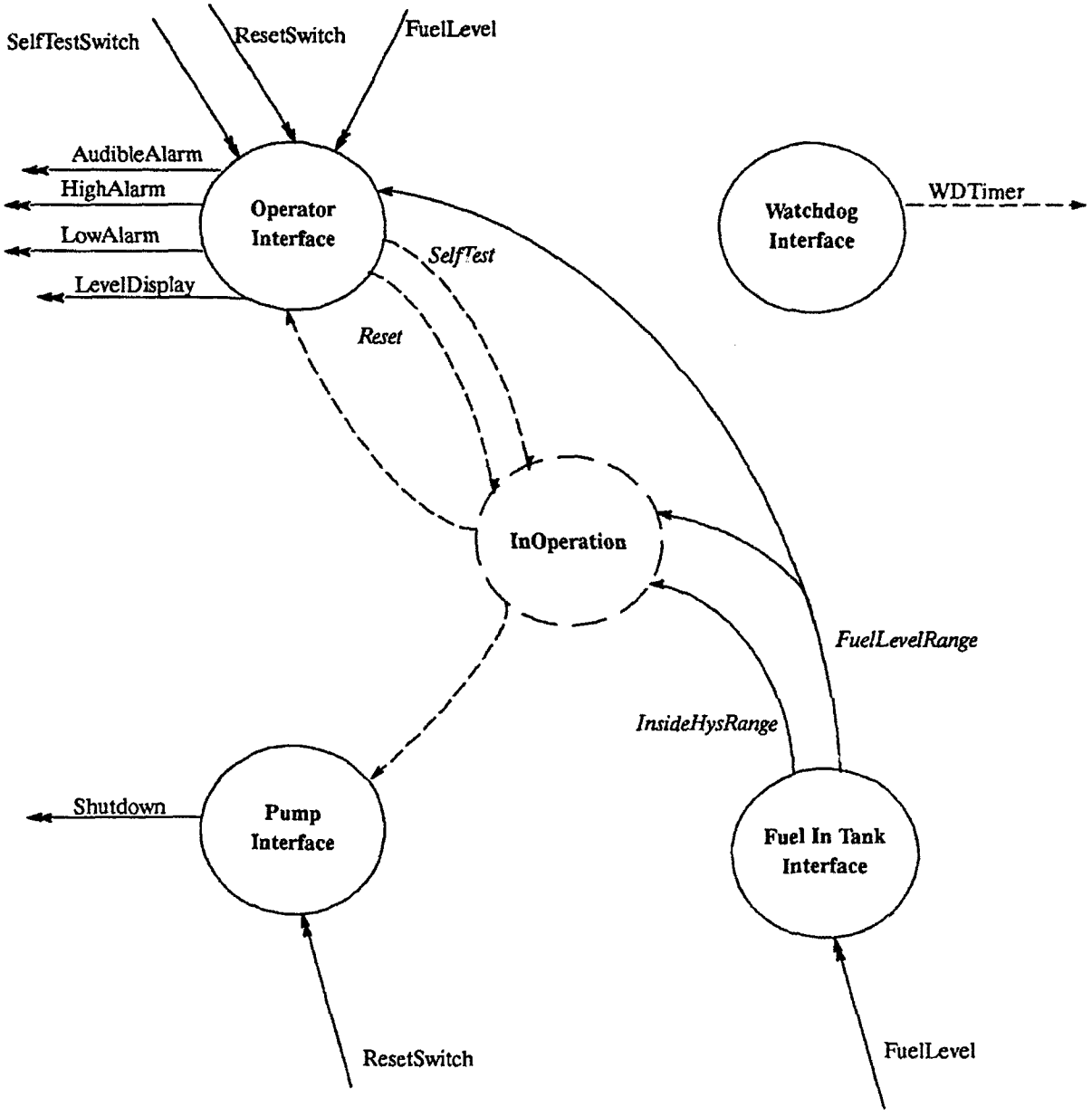

Figure 17. Fuel-Level Monitoring System: Transformation Diagram

B.5 INOPERATION OBJECT

B.5.1 Interface

The interface of the InOperation Object consists of four system modes, allowed transitions among them (but not the events that trigger the transitions), and several terms.

B.5.1.1 Modes

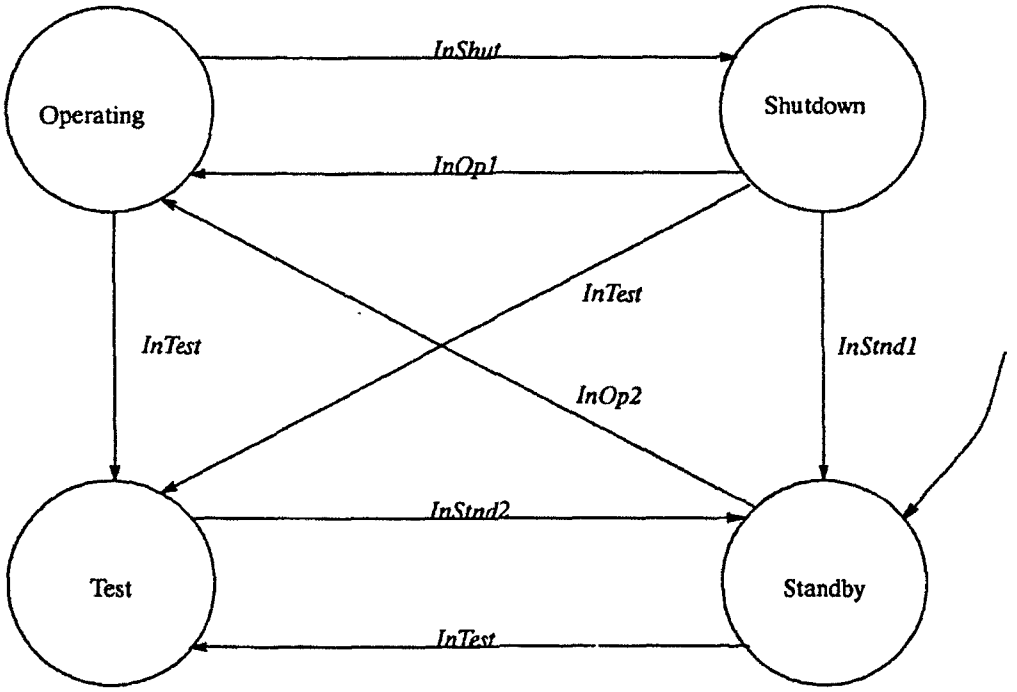

Figure 18. Fuel-Level Monitoring System: InOperation Modes

B.5.1.2 Terms

ShutdownLockTime :TIME:
= 0.2 s
(0 < *ShutdownLockTime* $\leq$ 0.2 s)

TestTime :TIME:
= DURATION (INMODE (Test))

B.5.2 Encapsulated Information

B.5.2.1 Events

InOp1 = @T(*InsideHysRange*) WHEN
(DURATION (INMODE (Shutdown)) < *ShutdownLockTime*)

InOp2 = @T(*Reset*) WHEN (*InsideHysRange*)

InShut = @F(*FuelLevelRange* = *WithinLimits*)

InStnd1 = @T(DURATION (INMODE (Shutdown)) > = *ShutdownLockTime*)

InStnd2 = @T(DURATION (INMODE (Test)) = 14s)

InTest = @T(*SelfTest*)

B.6 PUMP INTERFACE OBJECT

B.6.1 Interface

There is no externally usable information provided by this object.

B.6.2 Encapsulated Information

B.6.2.1 Controlled Variables

Name	Type/Values	Interpretation
Shutdown	:Boolean:	
	true	PumpSwitch = open
	false	PumpSwitch in {open, close}
PumpSwitch	:ENUMERATED:	
	closed	The contacts for switches for fuel pumps are closed, pumps running.
	open	The contacts for switches for fuel pumps are open, pumps off.

B.6.2.2 Terms

SwitchDelay :TIME: = 0.12 s

B.6.2.3 NAT Relation

Table 10. Behavior of PumpSwitch

Condition	Behavior
Shutdown = false and WDTimer > 0 and ResetSwitch = released	PumpSwitch = closed
Shutdown = true or WDTimer = 0 or ResetSwitch =pressed	PumpSwitch = open

B.6.2.4 Required Behavior

Table 11. Behavior of Shutdown

Mode	Condition	
Operating	INMODE	X
Shutdown	INMODE	X
Standby	X	INMODE
Test	X	INMODE
Shutdown =	false	true

Tolerance Delay: 0.05 s
Error: Not applicable

B.6.2.5 Output Data Items

Shutdown Signal

Acronym: ShutdownDevice

Hardware: Pump Shutdown Relay

Characteristics of Values:
Value Encodings: operate (1b)
shutdown (0b)

Data Transfer: PortC

Data Representation: Bit 1 of byte

B.6.2.6 OUT Relation

Table 12. Relations on Shutdown

Event	Action
@T(ShutdownDevice = shutdown)	Shutdown = true
@T(ShutdownDevice = operate)	Shutdown = false

Tolerance Delay: *SwitchDelay*
Error: Not applicable

B.7 WATCHDOG INTERFACE OBJECT

B.7.1 Interface

There is no externally usable information provided by this object.

B.7.2 Encapsulated Information

B.7.2.1 Controlled Variables

Name	Type/Values	Interpretation
WDTimer	:TIME:	
	00..MaxWDTime	:TIME: Time, in seconds (s) until the Watchdog system assumes the FLMS has failed.
MaxWDTime	:TIME: =0.9999 s	
TimerDelay	:TIME:	

B.7.2.2 NAT Relation

WDTimer. The Watchdog Timer is a decreasing function of time and counts down to 0 in *MaxWDTime* seconds. Its value is 0 when more than *MaxWDTime* seconds pass since the timer was set.

$$\text{LastPulseTime}(t) = \text{MAX}\{t_i : t_1 < t \text{ AND } \text{WDTimer}(t_i) = \textit{MaxWDTime}\}$$

Condition	Behavior
$t = t_0$	$\text{WDTimer}(t) = \textit{MaxWDTime}$
$t > t_0$	$\text{WDTimer}(t) = \text{MAX}\{0, \textit{LastPulseTime}(t) + \textit{MaxWDTime} - t\}$

B.7.2.3 Required Behavior

WDTimer. The timer should never be allowed to drop to 0.

Table 13. Behavior of WDTimer

Mode	Behavior
Any	INMODE
	WDTimer > 0

Tolerance Delay: 0.05 s
Error: Not applicable

B.7.2.4 Output Data Item

Computer Endurance Signal

Acronym: DogCmd

Hardware: Watchdog Timer

Characteristics of Values:
Value Encodings: pulsedog (41h)

Data Transfer: PORT (339h)

Data Representation: 8-bit unsigned integer

B.7.2.5 OUT Relation

Table 14. Relations on WDTimer

Event	WDTimer =
Watchdog.Pulse	*MaxWDTime*

Tolerance Delay: *TimerDelay*
Error: Not applicable

B.8 FUEL IN TANK INTERFACE OBJECT

B.8.1 Interface

The interface of the Fuel in Tank Interface Object consists of the monitored variable FuelLevel, the part of NAT that constrains the value of FuelLevel, and several terms that are defined on FuelLevel.

B.8.1.1 Monitored Variables

Name	Type/Values	Interpretation
FuelLevel	:LENGTH:	
	0.0..30.0	Level of fuel in the tank, in centimeters (cm), along the vertical axis on the left side of the tank, 5 cm from the front edge. The level is measured with respect to the scale. Figure 14 illustrates the pump and tank configuration.

B.8.1.2 NAT Relation

FuelLevel.

$$|d\,(\text{FuelLevel}\,(t))\,/\,dt| \leq \mathit{MaxLevelRate}$$

$$0.0\ \text{cm} \leq \text{FuelLevel} \leq 30.0\ \text{cm}$$

B.8.1.3 Terms

HighFuelLimit	:LENGTH: = 26.0 cm (*HighFuelLimit* > *LowFuelLimit*)
InsideHysRange	:BOOLEAN: = [(*LowFuelLimit* + *Hysteresis* < FuelLevel < (*HighFuelLimit* - *Hysteresis*)]
LevelHigh	:BOOLEAN: = FuelLevel > = *HighFuelLimit*
LevelLow	:BOOLEAN: = FuelLevel < = *LowFuelLimit*
LowFuelLimit	:LENGTH: = 14.0 cm (*LowFuelLimit* < *HighFuelLimit*)
FuelLevelRange	:ENUMERATED: LevelLow, WithinLimits, LevelHigh
WithinLimits	:BOOLEAN: = *LowFuelLimit* < FuelLevel < *HighFuelLimit*

B.8.2 Encapsulated Information

B.8.2.1 Terms

Hysteresis	:LENGTH: = 0.5 cm (0 < *Hysteresis* < (*HighFuelLimit* - *LowFuelLimit*))
k_1	:LENGTH: = 0.01891 × (*UpperCalibrationBound* - *LowerCalibrationBound*) cm
k_2	:LENGTH: = Delay × *MaxLevelRate*
LowerCalibrationBound	:LENGTH: = 13.0 cm (*LowerCalibrationBound* < *LowFuelLimit*)
MaxLevelRate	:LENGTH/TIME: = 0.375 cm/s
Offset	:LENGTH: = -0.01902 × (*UpperCalibrationBound* - *LowerCalibrationBound*) + *LowerCalibrationBound* cm
Scale	:LENGTH: = 1.03803 × (*UpperCalibrationBound* - *LowerCalibrationBound*) cm
UpperCalibrationBound	:LENGTH: = 27.0 cm (*UpperCalibrationBound* > *HighFuelLimit*)

B.8.2.2 Input Data Item

Differential Pressure.

Acronym: DiffPress

Hardware: Differential Pressure Unit

Characteristics of Values:

Values: DiffPress ∈ [0, 255]

Data Transfer: ADC (0)

Data Representation: 8-bit unsigned integer

Note: The differential pressure unit is calibrated with respect to the scale on the tank (Figure 14).

B.8.2.3 IN Relation

Table 15. Relations Between FuelLevel and DiffPress

Condition	DiffPress =
LowerCalibrationBound $\leq$ FuelLevel $\leq$ *UpperCalibrationBound*	((FuelLevel - *Offset*) / *Scale*) x 255
FuelLevel < *LowerCalibrationBound*	0
FuelLevel > *UpperCalibrationBound*	255
Device failure	0

Tolerance Delay: 0.2 s
Error: $k_1 + k_2$

B.9 OPERATOR INTERFACE OBJECT

B.9.1 INTERFACE

The interface of the Operator Interface Object consists of the monitored variables ResetSwitch and SelfTestSwitch and of events defined on them.

B.9.1.1 Monitored Variables

Name	Type/Values	Interpretation
ResetSwitch	:ENUMERATED:	
	pressed	The pushbutton labeled RESET is pressed.
	released	The pushbutton labeled RESET is not pressed.

B.9.1.2 Events

Reset :EVENT: @T(DURATION(ResetSwitch = pressed) > = *ResetDetected)*

SelfTest :EVENT: @T(DURATION(SelfTestSwitch = pressed) > = *SelftestDetected)*

B.9.2 Encapsulated Information

Figure 19 illustrates the detail breakdown of the Operator Object. There is one bubble for each controlled variable function encapsulated by the Operator Object.

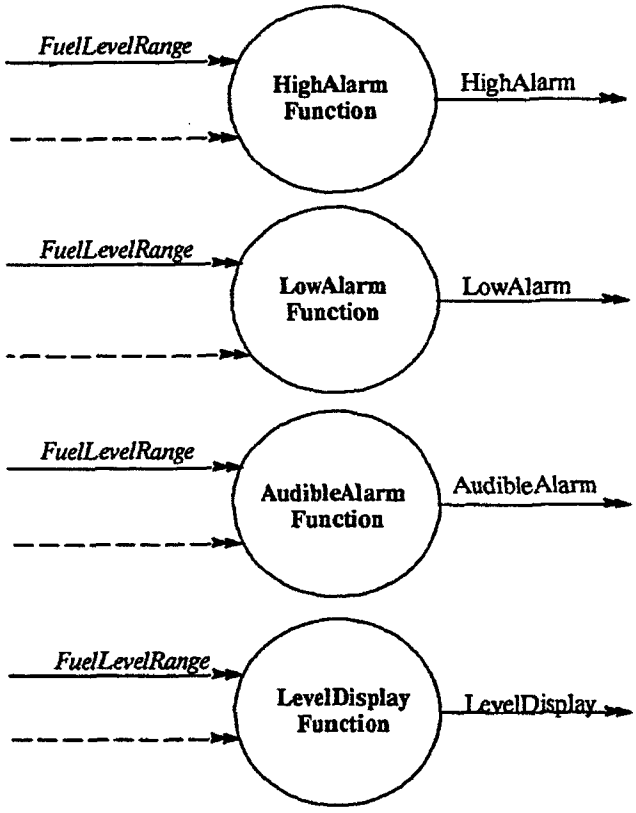

Figure 19. Operator Interface Object Transition Diagram

B.9.2.1 Monitored Variables

Name	Type/Values	Interpretation
SelfTestSwitch	:ENUMERATED:	
	pressed	The pushbutton labeled SLFTST is pressed.
	released	The pushbutton labeled SLFTST is not pressed.

B.9.2.2 Controlled Variables

Name	Type/Values	Interpretation
AudibleAlarm	:ALARM:	
	sound	The audible alarm is sounding.
	silent	The audible alarm is silent.
HighAlarm	:ALARM:	
	on	The alarm labeled FUEL LEVEL HIGH is on.
	off	The alarm labeled FUEL LEVEL HIGH is off.
LowAlarm	:ALARM:	
	on	The alarm labeled FUEL LEVEL LOW is on.
	off	The alarm labeled FUEL LEVEL LOW is off.
LevelDisplay	:LENGTH:	
	0.00..99.9	The value conveyed by the display labeled FUEL LEVEL.

B.9.2.3 Terms

AlarmDelay :TIME:

ButtonDelay :TIME:

Digit(x, k) :CHARACTER:

$$Digit(x,k) = \begin{cases} \dfrac{x \bmod 10^{k+1}}{10^k} & \text{if } \dfrac{x}{10^k} \neq 0 \\ \text{space} & \text{if } \dfrac{x}{10^k} = 0 \end{cases}$$

HighAlarmCol :INTEGER:
= 9

HighAlarmRow :INTEGER:
= 17

LevelDisplayRow :INTEGER:
= 6

LevelDisplayDigit(i) :INTEGER:

$$LevelDisplayDigit(i) = \begin{cases} 20 & \text{if } i = 0 \\ 18 & \text{if } i = 1 \\ 17 & \text{if } i = 2 \\ 16 & \text{if } i = 3 \end{cases}$$

LowAlarmCol	:INTEGER: = 29
LowAlarmRow	:INTEGER: = 17
MaxCol	:INTEGER: = 39
MaxRow	:INTEGER: = 24
MinCol	:INTEGER: = 0
MinRow	:INTEGER: = 0
P3	:LENGTH: = 0.1 cm
ResetDetected	:TIME: = 3 s
SelftestDetected	:TIME: = 0.5 s
SetDigit(x, i)	:EVENT: = @T(Character = *Digit(x, i)*) WHEN *CursorRow* = *LevelDisplayRow* AND *CursorCol* = *LevelDisplayDigit(i)*

B.9.2.4 Required Behavior

Alarm *AlarmName* state. The three alarms AudibleAlarm, HighAlarm, and LowAlarm are specified as instances of the class *AlarmName*. Table 16 specifies the behavior of the class. Table 17 specifies the definitions of the variables for instantiating each of the alarms.

Initial Value: (*AlarmName* = *InitialValue*), *SystemInit*

Table 16. Behavior of *AlarmName*

Mode	Triggering Event	
Operating	@F (*LimitsOK*)	@T (ENTER) WHEN *LimitsOK*
Shutdown	@F (*LimitsOK*)	X
Standby	X	X
Test	@T (*TestTime* $\geq$ *StartAlarmTime*)	@T (*TestTime* $\geq$ *EndAlarmTime*)
AlarmName =	*AlarmOn*	*AlarmOff*

Tolerance Delay: *ShutdownLockTime* / 2 - 0.001 s
Error: Not applicable

Table 17. Instance Alarm Definitions

Variable Name	Instances		
AlarmName	AudibleAlarm	HighAlarm	LowAlarm
Initial Value	*AlarmOff*	*AlarmOn*	*AlarmOn*
LimitsOK	*FuelLevelRange* = *WithinLimits*	NOT *FuelLevelRange* = *LevelHigh*	NOT *FuelLevelRange* = *LevelLow*
StartAlarmTime	0 s	0 s	2 s
EndAlarmTime	4 s	2 s	4 s
AlarmOn	sound	on	on
AlarmOff	silent	off	off

LevelDisplay.

Table 18. Behavior of LevelDisplay

Mode	Condition			
Operating OR Shutdown OR Standby	always			
Test		$0 \leq \text{Time} - t_0 < 4$	$4 \leq \text{Time} - t_0 < 14$	$14 \leq \text{Time} - t_0$
LevelDisplay =	Round(WaterLevel, - LOG(*P3*))	0.0	$\lfloor \text{Time} - t_0 \rfloor \times 11.1$	0.0

Tolerance Delay: 0.5 s
Error: 0.5 cm

B.9.2.5 Input Data Items

Reset.

Acronym: ResetDevice

Hardware: FLMS Push–button Array

Characteristics of Values: Values: on (1b)
off (0b)

Data Transfer: PortC

Data Representation: Bit 5 of byte

Self Test.

Acronym: SelfTestDevice

Hardware: FLMS Push–button Array

Characteristics of Values: Values: on (1b)
off (0b)

Data Transfer: PortC

Data Representation: Bit 7 of byte

B.9.2.6 IN Relation

Table 19. Relations Between ResetDevice and ResetSwitch

ResetDevice =	ResetSwitch =
pressed	on
released	off

Tolerance Delay: *ButtonDelay*
Error: Not applicable

Table 20. Relations Between SelfTestDevice and SelfTest

SelfTestDevice =	SelfTestSwitch =
pressed	on
released	off

Tolerance Delay: *ButtonDelay*
Error: Not applicable

B.9.2.7 Output Data Items

Cursor Position.

Acronym: CursorRow
CursorCol

Hardware: Console

Characteristics of Values:
Value Encodings: *MinRow* < = CursorRow < = *MaxRow*
MinCol < = CursorCol < = *MaxCol*

Data Transfer: SoftInt (10h), function 02h
CursorRow8088 register DH
CursorCol 8088 register DL

Data Representation: 8-bit unsigned integer

Screen.

Acronym: Character

Hardware: Console

Characteristics of Values:
Value Encodings:

x =	Value		x =	Value
space	32		vbar	179
.	46		ltee	180
0	48		lceil	191
...	...		rfloor	192
9	57		uptee	193
A	65		dtee	194
...	...		rtee	195
Z	90		hbar	196
a	97		cross	197
...	...		lfloor	217
z	122		rceil	218
bel	7		block	219

Data Transfer: SoftInt (10h), function 0Eh
8088 register AL

Data Representation: 8-bit unsigned integer

B.9.2.8 OUT Relation

Table 21. Relations on HighAlarm

Event	Action
@T(Character = space) WHEN CursorCol = *HighAlarmCol* and CursorRow = *HighAlarmRow*	HighAlarm = off
@T(Character = block) WHEN CursorCol = *HighAlarmCol* and CursorRow = *HighAlarmRow*	HighAlarm = on
@T(Character # space AND Character # block) WHEN CursorCol = *HighAlarmCol* AND CursorRow = *HighAlarmRow*	HighAlarm is undefined

Tolerance Delay: *AlarmDelay*
Error: Not applicable

Table 22. Relations on LowAlarm

Event	Action
@T(Character = space) WHEN CursorCol = *LowAlarmCol* and CursorRow = *LowAlarmRow*	LowAlarm = off
@T(Character = block) WHEN CursorCol = *LowAlarmCol* and CursorRow = *LowAlarmRow*	LowAlarm = on
@T(Character # space AND Character # block) WHEN CursorCol = *LowAlarmCol* and CursorRow = *LowAlarmRow*	LowAlarm is undefined

Tolerance Delay: *AlarmDelay*
Error: Not applicable

Table 23. Relations on AudibleAlarm

Event	Action
@T(Character = bel) WHEN *MinCol* < = CursorCol < = *MaxCol* and *MinRow* < = CursorRow < = *MaxRow*	AudibleAlarm = sound
@T(DURATION(Character # bel) > 0.5 s)	AudibleAlarm = silent

Tolerance Delay: *AlarmDelay*
Error: Not applicable

Table 24. Relations on LevelDisplay

Event	Action
SetDigit(X, i) for i in (0, 1, 2, 3)	LevelDisplay = X

Tolerance Delay: *LevelDelay*
Error: Not applicable

B.10 GLOSSARY

FLOOR (x : Real)	:Integer: Floor (x) = Max {i : Integer(i) and i < x}
LOG (x : Real)	:Real: 10**log(x) = x.
ROUND (x : Real)	:Real: Round (x) = Floor (x + 0.5)
SystemInit	:EVENT: = @F(t = t_0)
Time	:TIME: Time, in seconds, elapsed with respect to a fixed but arbitrary reference point t_0.

B.11 INDEXES

In the following indexes, a bold page number refers to where the item is defined. Other references are in italics.

B.11.1 Monitored State Variables

B.11.2 Controlled State Variables

B.11.3 Modes

B.11.4 Interface Terms

B.11.5 Miscellaneous Variables

This page intentionally left blank.

REFERENCES

Boehm, B. 1981 — *Software Engineering Economics*. Prentice-Hall.

Drusinsky, D. and D. Harel 1984 — *On the power of cooperative concurrency*, CS88–10. Department of Applied Mathematics, The Weizman Institute of Science.

Faulk, Stuart, James Kirby, Jr., Skip Osborne, D. Douglas Smith, and Steven Wartik 1991 — *Requirements Workshop Results Report*. SPC-91062-MC. Herndon, Virginia: Software Productivity Consortium.

Hammer, Michael, and Dennis McLeod 1981 — Database Description with SDM: A Semantic Database Model, *ACM Transactions on Database Systems*. 6:3.

Harel, D. 1984 — *Statecharts: A visual approach to complex systems*, CS84–05. Department of Applied Mathematics, The Weizman Institute of Science.

Harel, D., and A. Pnueli 1985 — On the Development of Reactive Systems. *Logics and Models of Concurrent Systems*. Edited by K.R. Apt. New York: Springer.

Heninger, Kathryn L. 1978 — Specifying Software Requirements for Complex Systems: New Techniques and Their Application. *IEEE TSE* SE6:2–13.

Jahanian, F., R. Lee, and A. Mok 1988 — Semantics of modechart in real time logic. *Proceedings of the 21st Hawaii International Conference on System Science.*

Mok, A.K. 1991 — Towards mechanization of real-time system design, to appear in *Foundations of Real-Time Computing: Formal Specification and Methods*. Kulwer Press.

Parnas, D., and J. Madey 1990 — *Functional Documentation for Computer Systems Engineering.* Queen's University Technical Report 90–287.

van Schouwen, A. J. 1990 — *The A–7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems.* Queen's University Technical Report 90–276.

Ward, P.
1989

The CASE Real-Time Curriculum. Software Development Concepts.

Zave, P., and W. Schell
1986

Salient features of an executable specification language and its environment, *IEEE TSE* SE-12, 2:312.

BIBLIOGRAPHY

Chmura, Louis J. and David M.Weiss, *The A-7E Software Requirements Document: Three Years of Change Data.*

Hester, S, D. Parnas and D. Utter, Using Documentation as a Software Design Medium. *The Bell System Technical Journal* 60:1941-1977, October 1981.

Mills, H.D., How to make exceptional performance dependable and manageable in software engineering. *Proceedings COMPSAC Conference* IEEE, October 17-31, 1980.

Parnas, D., G.J.K. Asmis, and J. Madey, *Assessment of Safety-Critical Software.* Queen's University Technical Report 90-295, December 1990.

This page intentionally left blank.